Dynamic Connector Synthesis: Principles, Methods, Tools and Assessment

Emil Andriescu, Amel Bennaceur, Paola Inverardi, Valerie Issarny, Romina Spalazzese, Roberto Speicys-Cardoso

CONNECT

Emergent Connectors for

Eternal Software Intensive Networked Systems

ICT FET IP Project

Deliverable D3.4

Dynamic Connector Synthesis: Principles, Methods, Tools and Assessment

SEVENTH FRAMEWORK PROGRAMME

http://www.connect-forever.eu

Inria · DOCOMO Euro-Labs · Ambientic · Lancaster University · Thales · Universita' dell'Aquila · technische universität dortmund · University of Oxford · Uppsala Universitet

Project Number	:	231167
Project Title	:	CONNECT – Emergent Connectors for Eternal Software Intensive Networked Systems
Deliverable Type	:	Report

Deliverable Number	:	D3.4
Title of Deliverable	:	Dynamic Connector Synthesis: Principles, Methods, Tools and Assessment
Nature of Deliverable	:	R
Dissemination Level	:	Public
Internal Version Number	:	3
Contractual Delivery Date	:	30 November 2012
Actual Delivery Date	:	17 December 2012
Contributing WPs	:	WP3
Editor(s)	:	Valérie Issarny and Amel Bennaceur (Inria)
Author(s)	:	Emil Andriescu (Ambientic - Inria), Amel Bennaceur (Inria), Paola Inverardi (UNIVAQ), Valérie Issarny (Inria), Romina Spalazzese (UNIVAQ), Roberto Speicys-Cardoso (Ambientic)
Reviewer(s)	:	Massimo Tivoli (UNIVAQ)

Abstract

CONNECT *adopts a revolutionary approach to the seamless networking of digital systems, that is, on-the-fly synthesis of the connectors via which networked systems communicate. Within* CONNECT*, the role of the WP3 work package is to devise automated and efficient approaches to the synthesis of such emergent connectors, provided the behavioral specification of the components to be connected. Thanks to WP3 scientific and technology development, emergent connectors can be synthesized on the fly as networked systems get discovered, implementing the necessary* mediation *between networked systems' protocols, from application down to middleware layers. This document being the final report about WP3 achievements, it outlines both: (i) specific contributions over the reporting period, and (ii) overall contributions in the area of automated, on-the-fly protocol mediation, from theory to supporting tool.*

Keyword List

Connectors, Protocol Mediators, Protocol Specification, Protocol Synthesis, Application-Layer Interoperability, Middleware-Layer Interoperability.

Document History

Version	Type of Change	Author(s)
0.1	Document outline	V. Issarny
0.2	Individual contributions on connector adaptation and theory	R. Spalazzese, P. Inveradi
0.3	Individual contributions on cross-layer mediation	E. Andriescu, R. Speicys, A. Bennaceur, V. Issarny
0.4	Individual contributions on on-the-fly connector synthesis	A. Bennaceur
1.0	Overall integration	A. Bennaceur
2.0	Completion and overall edition	V. Issarny
3.0	Final revision	V. Issarny

Document Review

Date	Version	Reviewer	Comment
Dec. 7	V2.0	M. Tivoli	Document review

Table of Contents

1 INTRODUCTION 7

2 FOURTH YEAR WORK 9
2.1 Third Review Recommendations 9
2.2 Contributions over the Period 10
2.2.1 Synthesis Enabler 10
2.2.2 Cross-layer Protocol Mediation 15
2.2.3 Connector Adaptation 16
2.2.4 Experiments 18

3 CONNECTOR SYNTHESIS: FROM THEORY TO PRACTICE 21
3.1 The Connect Theory of Mediators 21
3.1.1 Ontology-based Networked System Model 21
3.1.2 The Theory of Mediator in a Nutshell 22
3.2 The Connect Synthesis Enabler 23
3.2.1 Mapping Interfaces 23
3.2.2 Synthesizing Abstract Mediators 25
3.3 Beyond Connect Architecture: Toward a Mediator Synthesis Service 26

4 ASSESSMENT 29

5 CONCLUSION 37

APPENDIX: PAPERS ON CONNECTOR SYNTHESIS 39

A AUTOMATED SYNTHESIS OF MEDIATORS TO SUPPORT COMPONENT INTEROPERABILITY 41

B AN AUTOMATED MEDIATION FRAMEWORK FOR CROSS-LAYER PROTOCOL INTEROPERABILITY 77

C SYNTHESIZING CONNECTORS MEETING FUNCTIONAL AND PERFORMANCE CONCERNS 91

D ACHIEVING INTEROPERABILITY THROUGH SEMANTICS-BASED TECHNOLOGIES: THE INSTANT MESSAGING CASE 101

BIBLIOGRAPHY 113

1 Introduction

A core challenge of CONNECT is to provide a solution to the dynamic synthesis of CONNECTors, *aka* emergent middleware, so as to make interoperable functionally compatible *Networked Systems* (NSs) given their behavioral semantics. As a result, highly heterogeneous, but functionally compatible, NSs may successfully coordinate despite discrepancies, or mismatches, in the protocols they use for interacting with their environment. Within CONNECT, we address protocol mismatches from the application down to the middleware layers, while we assume that NSs interact over IP.

On the one hand, middleware provide services that facilitate the connection of networked systems despite the heterogeneity of the underlying platforms, operating systems, and programming languages. Specifically, middleware introduce specific message formats and coordination models, which makes it difficult (or even impossible) for applications using different middleware to interoperate: an application implemented on top of CORBA cannot communicate with an application developed using SOAP. Furthermore, the evolving application requirements lead to a continuous update of existing middleware tools and the emergence of new approaches. For example, SOAP has long been the protocol of choice to interface Web services but RESTful Web services [45] are becoming increasingly common today. As a result, application developers have to juggle with a myriad of technologies and tools, and to include *ad hoc* glue code whenever it is necessary to compose applications implemented using different middleware. Solutions to middleware-layer interoperability facilitate this task [61], either by providing an infrastructure to translate messages into a common intermediary protocol such as in the case of Enterprise Service Buses [35], or by proposing a Domain Specific Language (DSL) to describe the translation logic and to generate corresponding bridges [27]. These solutions, however, provide only an execution framework and still require developers to create or specify the translations needed to make applications interoperate.

On the other hand, solutions to application-layer interoperability target higher automation and loose coupling. In particular, they rely on intermediary entities, *mediators* [115], to enforce interoperability by mapping the interfaces of software components and coordinating their behaviors. There are many approaches to generate mediators based on an *interface mapping* [119, 91]. Interface mapping (also called adaptation contract [24, 84]) establishes the semantic correspondence between operations and data of the considered components. The computation of interface mappings may then be automated through either measuring the syntactic similarity of messages [91] or reasoning about the semantic annotations of the interfaces using ontologies. Ontologies build upon a sound logic theory to enable reasoning about the domain based on an explicit description of domain knowledge as a set of axioms [6]. Furthermore, application interoperability solutions not only consider the semantics of operations and data but also the behavioral semantics of components, which makes the reasoning about interoperability more accurate and the generation of mediators more amenable to automation. They all assume, however, the use of the same underlying middleware (e.g., SOAP) and focus on dealing with mediation at the application layer only.

Hence, even though there exist many interoperability solutions, they are inappropriate for one of the two following reasons: either (i) they deal with application heterogeneity and generate corresponding mediators but fail to deploy them on top of heterogeneous middleware, or (ii) deal with middleware heterogeneity while assuming the same application atop and rely on developers to provide all the translations that need to be made. Within CONNECT, we have been arguing that seamless interoperation is a cross-cutting concern and interoperability solutions must consider conjointly application and middleware layers:

- The application layer provides the appropriate level of abstraction to reason about interoperability and automate the generation of mediators;
- The middleware layer offers the necessary services for realizing the mediation by selecting and instantiating the specific data structures and protocols.

The development, from theory to practice, of a comprehensive approach to mediator synthesis reconciling application and middleware-layer protocols so as to make interoperable heterogeneous networked systems that implement *compatible* functionalities, has been the focus of the CONNECT Work Package WP3. Still, we should highlight the close relation of WP3 work with the one undertaken within WP1, especially regarding the mediator engine, which is part of the deployment enabler. As this document is the final report about WP3 work, it provides an overview of the work carried out during the final year of the project

as well as of the key overall achievements of WP3. In addition, detail about the scientific contributions of the work package may be found in the papers provided in the appendix.

Precisely, this document is structured as follows:

- Chapter 2 provides an overview of the work undertaken during the reporting period, which relates to: (i) implementing the recommendations provided following the project's third review, and (ii) meeting the key objective of WP3 that is to develop automatic CONNECTor synthesis approaches that can be efficiently performed at runtime.
- Chapter 3 provides an overview of the CONNECT contributions in the area of automated mediator synthesis, from theory to practice.
- Chapter 4 assesses the CONNECT approach to automated CONNECTor synthesis against the initial objectives and in particular the following assessment criteria that are set in the project's Description of Work:
  - Kinds of mismatches that can be prevented/solved;
  - Capability of taking into account high-level non-functional properties;
  - Compositionality that implies efficiency, dynamicity, and hence evolution;
  - Architectural connector patterns that can be supported;
  - Automation degree of the CONNECTor model construction;
  - Automation degree of the CONNECTor's actual code development;
- Chapter 5 concludes this document with a summary of our key contributions in the area of automated mediator synthesis together with open research questions that remain.
- The Appendix collates a number of papers produced by the CONNECT consortium in the final year in the area of automated mediator synthesis, most of which are still at the under publication stage. These papers detail the work undertaken during the final year, while at the same time provide a comprehensive survey of our overall approach to CONNECTor synthesis together with its application in real world use cases.

2 Fourth Year Work

This chapter provides a brief overview of the WP3 work over the final reporting period, while further detail may be found in the papers of the appendix. We first survey the WP3-related recommendations that were stated after the third project review.

2.1 Third Review Recommendations

Recommendations for WP3 work after the third review were addressed as follows:

- *The work package has provided a detailed analysis and comparison of the two mediator synthesis approaches. In finalizing the work, the reviewers recommend to provide some guidelines as to when these two complementary or competing approaches should be employed. It may even be the case that they can be integrated into a single tool*?

  The two approaches to CONNECTor synthesis, i.e., mapping- and goal-based, have been integrated into a single solution. Specifically, we decided to concentrate on finalizing the implementation of a synthesis enabler leveraging the mapping-based synthesis approach, which was the most general and the most amenable to full automation. The resulting synthesis enabler handles N-M interface mappings (i.e., N operations of one networked system map to M operations of the peer networked systems) and ambiguous mappings (One sequence of operations of one NS may map to multiple sequences of operations of another NSs). In addition, the enabler is parameterized with an optional goal that specifies the target specialization for the connection. – See Section 2.2.1 for more detail –.

- *Concerning the earlier comment (under WP2) on the inconsistencies between the synthesis approaches of WP2 and WP3, it looks like the framework of WP3 implements a different notion of connector compatibility, based on the set of traces defined by the to be connected eLTS: Definition 9 states (and Def. 23 reformulates that a bit differently) that two eLTS are compatible iff there exists at least one pair of executions that can be jointly executed by a connector (with finite memory). This is not sufficient, unless all executions are essentially equivalent. One would expect at least a universal quantification over the behaviors of the environment.*

  The theory of mediators whose final version was detailed in Deliverable D3.3 considers a different notion of connector compatibility than that of the theory of CONNECTors developed in WP2. Precisely, the WP3 theory of mediators allows for NSs to connect despite possible behavioral mismatches under some systems runs, while the WP2 theory is more conservative and prevent any behavioral mismatch occurrence for the connection of networked systems. Practically, the WP2 apporach is more sustainable in the context of automated connections of systems where we want to minimize user intervention. As a result, the concrete approach to CONNECTor synthesis that is implemented by the dedicated enabler also adopts a conservative approach similar to that of WP2 theory and hence have a restrictive implementation of WP3 theory of mediators. – See Section 2.2.1 for more detail –.

- *In the list of acronyms (p.7) abbreviation eLTS is used for both enhanced and extended LTS. Are both terms necessary or is this a mistake*?.

  This is indeed a mistake and "*Enhanced LTS*" should have been used throughout.

- *The non-distinction of inputs and outputs in the approach of WP3 has its limits: when m and $\overline{m}$ are treated symmetrically, it is impossible to formulate the (natural) requirement that in a closed system, unmatched outputs are allowed whereas unmatched inputs are not.*

  This is due to the simplified definition of parallel composition for the sake of brevity, while the definition provided in D3.3 was emphasizing the adoption of CCS-like parallel composition. In practice, the approach to CONNECTor synthesis, from theory to supporting enabler, does distinguish between input and output parameters. Hence, the stated natural requirement is indeed addressed.

- *In a protocol composition (Def. 4) the union of final states of components are not states of the product. Probably what is meant is the product of final states.*

This is indeed a mistake, what is meant is the product of final states. More precisely the set of final states of the product is the product of final states of the two networked systems.

2.2 Contributions over the Period

Whereas the definition of the theory of mediators was finalized in the previous reporting period (see Deliverable D3.3 for detail), our work during this period has been primarily focused on the finalization of the CONNECT synthesis enabler so as to support the automated synthesis of CONNECTors enabling functionally compatible systems to interoperate despite behavioral mismatches at the upper application layer and/or lower middleware layers.

The following sections survey our fourth year achievements, which are further detailed in related publications provided as appendices. Achievements relate to:

1. Finalization of the CONNECT synthesis enabler, which is integrated in the overall CONNECT architecture that is assembled within WP1 (See Deliverable D1.4 for a comprehensive presentation of the final CONNECT architecture) and experimented within WP6 (See Deliverable D6.4 for an overview of the experimental work undertaken in the fourth year and related evaluation of CONNECT architecture and enablers). Overview of the synthesis enabler is given in Section 2.2.1.

2. The CONNECT synthesis enabler generates an abstract mediator in the form of an enhanced LTS that coordinates the actions produced and consumed by the networked systems that get connected, through the addition of the necessary translation and buffering of actions, assuming synchronous interactions. The execution of any abstract mediator then relies upon the deployment of the mediator on the Starlink engine where the enhanced LTS is translated into relevant colored automata that parse (resp. compose) messages issued by (resp. to) the middleware layer to translate them into abstract actions coordinated by the enhanced LTS. The key role of Starlink in the CONNECT architecture is detailed in WP1 deliverables. However, Starlink is a general-purpose mediator engine that requires the specification of middleware messages, which may quickly become cumbersome despite the introduction of a dedicated Domain Specific Language. We have thus studied a solution to the synthesis of message parsers and composers that allows composing and thereby reusing legacy implementations of such message parsers/composers. The proposed approach further tackles the complex issue of cross-layer interoperability because in real world scenarios the boundaries between both layers tend to disappear, mixing application and middleware data across multiple layers. This confusion is caused by multiple factors such as performance optimizations, simplified development or bad design decisions. Cross-layer protocol mediation is further discussed in Section 2.2.2.

3. Adaptation of CONNECTors has been put forwards as one of the research challenges for CONNECT. However, the automated synthesis of CONNECTors, from theory to supporting enabler, has remained the core focus of WP3 due to the inherent complexity of the target objective, especially when it goes with delivering a functioning prototype to be integrated in the overall CONNECT architecture. Still, during the fourth year and in collaboration with WP5, we have started investigating the adaptation of CONNECTors regarding the enforcement of non-functional properties, which is discussed in Section 2.2.3. Adaptation of deployed CONNECTors has further been investigated as part of WP1 and the interested reader is referred to Deliverable D1.4 for detail.

4. CONNECT enablers have been extensively experimented with, as part of WP6 activities considering applications of the GMES area and of the mobile environment. Additional experiments have also been undertaken to assess specific enablers as is the case with the synthesis enabler. Experiments with the synthesis enabler are outlined in Section 2.2.4.

2.2.1 Synthesis Enabler

Given two Networked Systems (NSs) that are functionally compatible (i.e., one requires an affordance that semantically matches an affordance provided by the other), the CONNECT synthesis enabler either

generates a mediator that allows the two NSs to collaborate despite behavioral mismatches related to the implementation of heterogeneous interfaces and/or coordination protocols from application down to middleware layers, or specifies that such a mediator can not be automatically generated -if one exists.

The synthesis enabler specifically relies on the provision of a semantically-annotated system model for each NS. As presented in Deliverable D1.4 on CONNECT architecture, the required NS model that derives from the theory of mediators (see Chapter 3) decomposes into:

(i) A reference to the domain ontology used for the model definition.

(ii) The semantically-annotated interface of the NS that sets the signatures of the operations (*aka* actions) provided and required by the NS, and associated ontology concepts.

(iii) The affordances provided and required by the NS where each affordance comes along with its behavioral specification in terms of enhanced LTS over the (abstracted) operations of the NSs. In particular, the actions of the enhanced LTS abstract the underlying middleware protocol used for remote interaction, by translating any remote operation into either an input or an output action according to the semantics of the underlying middleware (see Deliverable D3.2).

As detailed in Deliverables of WP1, we can reasonably assume to have such NS models available, possibly with the assistance of machine learning technologies. One may further note that the above model is similar to that put forward by ontologies for semantic Web services.

Mapping-based Mediator Synthesis

Given the above models of two NSs implementing functionally matching affordances, the CONNECT synthesis enabler combines ontology reasoning and constraint programming to generate a mapping between the interfaces of the NSs. The generated interface mapping is then taken as input to automatically synthesize a mediator that ensures the safe interaction of the NSs to realize the given affordance, i.e., it ensures deadlock-freedom. We qualify the resulting synthesis enabler as *mapping-based*.

Next to mapping-based mediator synthesis, we investigated goal-based mediator synthesis in the third year of the CONNECT project. However, the approach turned to be less suited for full automation due to its reliance on common language inference and reasoning about matching per pair of traces. We have then focused on complementing the mapping-based CONNECTor synthesis with goal-based connection.

First version of the mapping-based synthesis enabler was introduced last year in Deliverable D3.3 and its early integration within the CONNECT architecture was also tackled as part of WP1 work (see Deliverable D1.3). However, the proposed synthesis enabler had to be enhanced so as to deal with N-M action mappings and with ambiguous action mappings. We have also accounted for the possible specification of goals for the specialization of target connections.

Figure 2.1 depicts the resulting synthesis enabler, which is made up of three modules whose design and implementation are detailed in Appendix A:

1. The *ontology encoding module* (see Figure 2.1-❶) classifies the ontology using the Pellet reasoner[1] that is an open-source java library for OWL DL reasoning. Then, the module encodes the classified ontology into a map that associates each concept with a bit vector.

2. The *interface mapping module* (see Figure 2.1-❷) uses Choco[2] to compute the mapping between the interfaces of the components given as input. Choco is an open-source java library for constraint solving and constraint programming. It is built on an event-based propagation mechanism with backtrackable structures. Choco does not manage ontology relations such as subsumption but, thanks to the above ontology encoding using bit vectors and the associated modeling of constraints, we are able to specify interface mapping as a constrained optimization problem with operators supported by the Choco library.

[1] http://clarkparsia.com/pellet/
[2] http://choco.emn.fr/

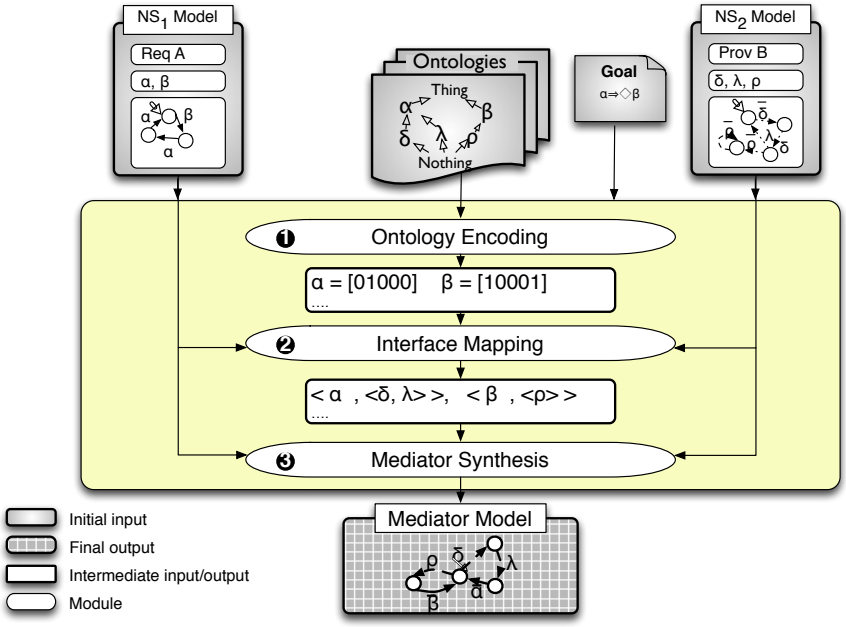

Figure 2.1: The CONNECT Enabler for Mediator Synthesis

3. The *mediator synthesis module* (see Figure 2.1-❸) relies on the generated mappings to synthesize the mediator M, which coordinates the processes P_1 and P_2 that represent the respective behavior of the NS affordances as enhanced LTSs. The mediator is incrementally built by forcing the two processes P_1 and P_2 to progress consistently so that if one requires the sequence of actions X_1, the interacting process is ready to engage in a sequence of provided actions X_2 to which X_1 maps. Given that an interface mapping guarantees the semantic compatibility between the actions of the two components, then the mediator synchronizes with both protocols and compensates for the differences between their actions by performing the necessary transformations. The mediator further consumes the extra output actions so as to allow protocols to progress.

Dealing with Goals

As outlined above, the mapping-based synthesis of a mediator ensures that the two connected networked systems will successfully coordinate *via* the mediator as long as they implement the behavior specified for the given affordance. We are currently extending the implementation of the synthesis enabler with goal specification. As presented in Deliverables D1.3 and D1.4, the behavior of the affordance may additionally be customized by the specific intent, or goal, for the connection. In this case, still assuming that P_1 and P_2 are the enhanced LTSs representing the behavior of the affordances of the NSs to be connected, and G is the Linear Temporal Logic (LTL) formula specifying the goal, we have to synthesize a mediator model M such that the composition $P_1 \| M \| P_2$ reaches a final state and also satisfies the goal, i.e., $P_1 \| M \| P_2 \models G$. The idea is then to retain only the behavior (of the affordance) of each networked system, P_1 and P_2, that satisfy the goal property. We denote the resulting processes as P_1' and P_2'. This computation is performed by calculating the intersection between the enhanced LTS representing the behavior of the networked system and the automaton representing the goal. When the goal represents safety properties, it can be represented as a finite state automata. When the goal represents liveness properties, it needs to be represented as a Büchi automaton and the enhanced LTSs of the networked systems are required to be total, i.e., they have an outgoing transition in each state. By doing so, we can prove that the resulting mediator also verifies the goal using verification via projection [74]. Indeed, M can be projected on P_1' or P_2', which both verify G, and hence M also verifies G.

Consider the example in Figure 2.2 where P_1 represents the behavior of a client application that first authenticates using a username and a password. The client chooses to either create an order or to cancel an existing one. If he creates an order, he adds as many items as he wants to this order and

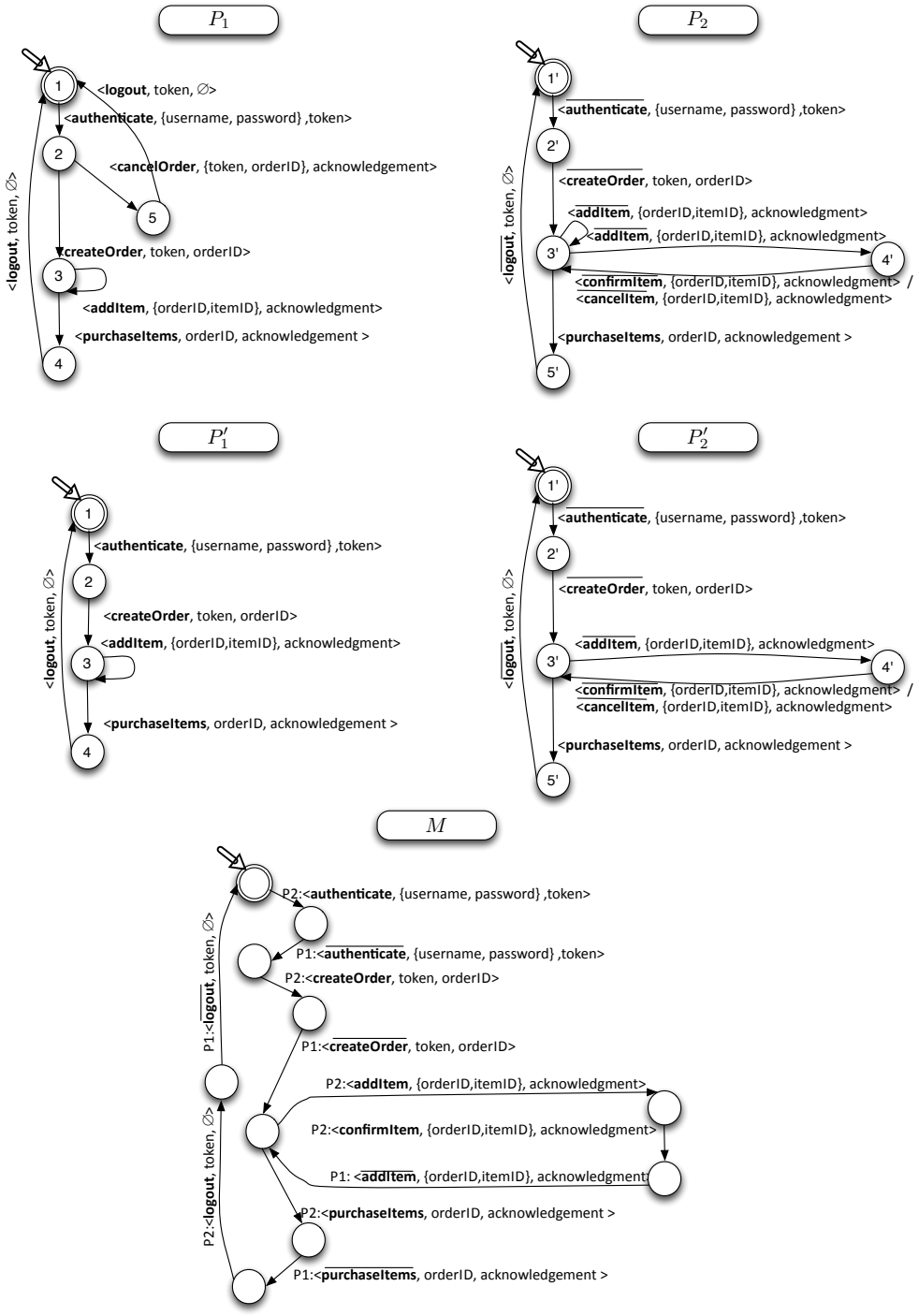

Figure 2.2: Example of a Goal Restricting the Behavior of the CONNECTed System

then completes the purchase. In both cases, i.e., purchase or a cancelation, he logs out to terminate. P_2 represents the behavior of a server-side application that expects the client to authenticate and create an order. The server allows the client to either add items to the order or to add and confirm the order. Then, he can validate the order and purchase the items. The mapping-based synthesis would not be able to generate a mediator between P_1 and P_2 since P_1 requires an action, cancelOrder, that is not supported by P_2. Let us now consider that the goal is to eventually purchase an item and to guarantee that the addition of an item is followed by a confirmation, i.e., $G = \Box\,(\mathsf{addItem} \Rightarrow \Diamond\ \mathsf{confirmItem}) \wedge \Diamond\ \mathsf{purchaseItems}$. The goal in this case constrains the behavior of the networked systems P_1 and P_2. The first step is to compute the intersection of the behavior of each networked system with the goal property, resulting in two enhanced LTSs P_1' and P_2'. Then, we rely on the mapping-based synthesis approach to generate a mediator between P_1' and P_2'. The mapping-based synthesis approach is able to generate the mediator M between P_1' and P_2', which further satisfies the goal.

From Abstract Mediator to Emergent Middleware

Once the mediator model is generated, it needs to be refined and deployed into a concrete artefact so as to realise the specified translations and coordination. This artefact is called an *emergent middleware* [19]. The emergent middleware refines the mediator model by incorporating information about underlying middleware and network layers. In particular, the emergent middleware:

(i) Intercepts the input messages,

(ii) Parses them in order to abstract from the communication details and represent them in terms of actions as expected by the mediator,

(iii) Performs the necessary data transformations, and

(iv) Uses the transformed data to construct an output message in the format expected by the interacting component.

As detailed in Deliverable D1.4, Steps (i), (ii) and (iv) are performed using middleware-specific parsers and composers. We can either use existing middleware libraries to perform this task or rely on an interpretation framework such as Starlink (see WP1 deliverables) to generate them at runtime. Step (iii) needs further computation. Even though ontological subsumption guarantees the semantic compatibility between concepts, we still need to specify the necessary data transformations in order for the mediator to deal with the syntactic difference between the input/output data.

Data mapping is a large and complex problem space [103]. Nevertheless, we should distinguish two cases. In the case of simple types only, the translation is quite straightforward and often consists in simple cast operations. However, in most cases we need to deal with complex data types, e.g., mapping two elaborated XML Schemas. Since the focus of our work is on the dynamic synthesis of mediator rather than a novel approach for data transformations, we rely on existing approaches for data mapping that devise different techniques to infer the transformations needed to translate from an XML Schema to another. We refer the interested reader to the complete survey by Shvaiko and Euzenat [103] for a thorough survey and analysis of the approaches that can be applied with this goal in mind. In particular, we rely on the Harmony library[3] to compute the matching between heterogeneous XML Schema and use Apache Dozer[4] to execute the sequence of functions that translates an instance of the source schema into a valid instance for the target schema.

Synthesis Enabler Prototype

The final version of the CONNECT synthesis enabler is now available for download from the software page of the CONNECT Web site at `https://www.connect-forever.eu/software.html`. In addition, detailed presentation of the proposed mapping-based mediator synthesis is provided in Appendix A on "*Automated Synthesis of Mediators to Support Component Interoperability*", which covers design, implementation and experiment-based assessment of the solution.

[3] `http://openii.sourceforge.net/index.php?act=tools&page=harmony`
[4] `http://dozer.sourceforge.net/`

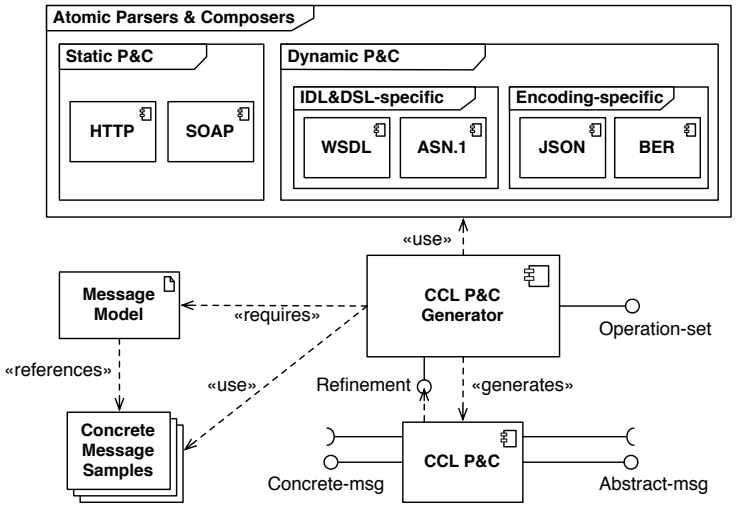

Figure 2.3: Synthesis of Parsers and Composers

2.2.2 Cross-layer Protocol Mediation

As discussed in the previous section, whenever NSs must communicate using heterogeneous message formats, *Parsers* and *Composers* (P&C for short) are required to interpret messages from one NS and to generate messages in a format that the other NS understands. Most approaches, including Starlink, that enable processing heterogeneous message formats require the specification of P&Cs using a supporting DSL, e.g., ABNF[5][46, 29, 23]. This technique presents three main weaknesses:

1. The workload required for the specification becomes overwhelming when protocols present complex message structures,
2. All message encapsulation layers are handled at once (thus restraining the reuse of parsers and composers), and
3. It is limited by the expressiveness and suitability of the DSL with respect to the type of encoding (e.g., text, XML, BER).

We have thus investigated a new approach to the synthesis of message P&Cs, which is better suited for the complexity of messages exchanged in real world applications. The approach lies in the synthesis of *Composite Cross-Layer (CCL) parsers and composers* whose goals are:

(i) To make it easier to automatically generate message P&Cs for applications using heterogeneous middleware,

(ii) To provide the mediator with an abstract representation of the protocol data that is independent of the specific middleware and application layers used by the interacting NSs, and

(iii) To annotate application data with parsing-related information that is useful to ensure proper mediation.

Provided with an abstract representation of application data, mediators can focus on how to enable NS to interoperate instead of dealing with low level issues such as data encapsulation and communication.

As depicted in Fig. 2.3, the *CCL P&C Generator* produces Cross-Layer Parsers and Composers (*CCL P&C*) at design time based on three inputs:

(i) A set of *Atomic P&C*s, stored in a repository, that transform a specific data representation format into a uniform parse-tree representation.

[5]Augmented Backus-Naur Form: http://www.ietf.org/rfc/rfc2234.txt

(ii) A *Message Model* that defines the strategy for assembling *Atomic P&Cs* in order to deal with multiple data encapsulation layers. Indeed, protocol messages usually contain data scattered over multiple encapsulation layers, each using a different encoding format based on a different standard (e.g., SOAP over HTTP). Thus, in general, to have access to the overall message data, different parsers and composers are required. To reduce the effort of generating P&Cs for application messages, our approach relies on modular chaining and re-use of legacy P&Cs as specified by the message models. This increases the flexibility, and reduces the effort, of composite P&Cs generation for specific application messages.

In addition, cross-layer data dependencies between message encapsulations are a result of complex interactions among protocol layers. As a result, software components may include application data into middleware messages, or add middleware data to the stack of another middleware, further complicating NS interoperation. To provide mediators with an abstract representation of application messages, parsed data must be annotated so that they can be transferred to mediators even if they are mixed with middleware data. Annotation of data relies on the adequate specification of the message model.

(iii) A set of *Concrete Message Samples* that is optional, allows the engine to refine the generated parser with additional rules discovered from actual messages.

Our solution to the synthesis of composite Cross-Layer Parsers and Composers is detailed in Appendix B on "*An Automated Mediation Framework for Cross-Layer Protocol Interoperability*". The paper includes a description of the supporting software prototype and details integration with mapping-based mediator synthesis.

2.2.3 Connector Adaptation

In addition to finalizing the CONNECT synthesis enabler, we have investigated the *adaptation* of CONNECTors. While run-time reconfiguration of CONNECTors has been studied within WP1, we have studied the adaptation of CONNECTors with respect to both functional and non-functional properties within WP3, acknowledging that the latter is joint work with WP5.

Functional Adaptation

The mediator we synthesize enables the networked systems to interoperate and achieve the desired goal (if it is specified) assuming that the models of these systems and the associated domain ontology are correct (i.e., the abstract system model is compliant with the actual system interface and behavior, and the ontology defines the necessary concepts and relevant relations among them) although they may provide only partial knowledge about the NSs and/or domain. However, even though a generated mediator may be valid at a given time, it may become invalid as the context evolves. Functional adaptation aims at preserving/re-establishing a mediator model that gets invalidated because of some uncontrolled changes either in the networked systems or their models. To that end, we have been exploring approaches for dynamic-adaptive systems and how these approaches can be applied to deal with the critical issue of incomplete or imprecise knowledge when enforcing interoperability. In the context on CONNECT, we identified the following cases:

1. *Changes of the learned behavioral model of a networked system.* The change might be due either to an evolution of the networked system or to imprecision in the learning algorithm. Since the interface remains unchanged in this case, only the behavior analysis need to be resumed. Furthermore, an interesting property of the learning algorithm devised in CONNECT, and which is based on the L* algorithm, is its monotonicity [18]. Indeed, it can only refine or add transitions. As a result, the synthesis does not have to be resumed from scratch, rather it can be pursued from the state where the new transition has been introduced. Nevertheless, we need to distinguish networked system evolution, which need to restart learning from scratch, from updates on the learning model. One way to make sure that the change is not due to system evolution is to check if the traces accepted before are still accepted. If they are not valid anymore, we can state that a change has been made. Otherwise, we cannot state anything.

2. *Changes of the interface of a networked system.* In this case, the behavioral model of the networked system has also to change and we need to re-compute the interface mapping. The computation may be incremental if the change consists in addition of actions but needs to start from scratch otherwise. Nevertheless, we do not have to encode the ontology, which is the most time-consuming step in the synthesis.

3. *Changes of the ontology.* In this case, the synthesis has to be started from scratch since the computed mappings may not hold anymore.

As outlined above, adaptation of a CONNECTor following changes in the functional properties of the connected NSs relies on applying part or whole of the mediator synthesis process, depending on the specific changes. This further goes along with reconfiguring the resulting emergent middleware, as discussed in Deliverable D1.4.

Non-functional Adaptation

Changes to a connector may also be due to non-functional concerns, for which we studied adaptation from two perspectives:

1. One stream of work is described in Appendix C on "*Synthesizing Connectors Meeting Functional and Performance Concerns*". Our focus is on pre-deployment time when heterogeneous NSs with some performance requirement, trigger the need of a CONNECTor. Our aim is then to produce a CONNECTor satisfying both the functional and performance required characteristics by exploiting adaptations that may arise from: (i) alternative NS behaviors implementing the same functionality (alternative behaviors slicing), (ii) number of loop iterations (tuning the upper bound of number of loop iterations), and (iii) possible deployment configurations (the most convenient deployment configuration). The proposed approach produces an intermediate CONNECTor and, by considering analysis-based strategies acting on the three variation points mentioned above, suggests either how to properly prune the CONNECTor behavior or how to deploy it in order to improve the connected system performances to target performance requirements where the term *connected system* means the one composed by the NSs plus the synthesized mediator. The main contributions of this work are: (a) the definition of the process combining the CONNECTor synthesis and the analysis-based reasoning on it to meet non functional requirements on the whole connected system, (b) the identification of three general strategies to improve the connected system performance, and (c) an implementation of the performance analysis-based reasoning step.

2. The other work stream, developed in collaboration with WP5 is described in [17] and [51], and is more detailed in WP5 deliverable.

   Paper [17] reports on a first study to set an adaptation approach involving the synthesis, DePer and monitoring enablers. A cycle among the three enablers has been identified that: starts with the automated synthesis of a mediator enabling the functional interoperation among heterogenous NSs (Synthesis); is supported by pre-deployment assessment applying stochastic model-based analysis to assess the desired non functional properties and to take appropriate design decisions by providing *a priori* feedback about how the system is expected to operate (DePer), and is supported by lightweight flexible monitoring infrastructure that observes the run-time CONNECTor behavior to provide feedback to support run-time adaptation.

   The work in [51] is then mainly focused on the collaboration between DePer and synthesis enablers. It takes place at both pre-deployment time and run-time and its aim is to cope with problems arising from the execution environment due to the uncertainties about the knowledge of the environment at pre-deployment time and the evolution of the context. By reasoning about systems' specification, during the pre-deployment phase the approach produces a mediator that satisfies the functional, performance and dependability requirements. At run-time when a performance or dependability violation notification is received, the approach by reasoning about the new specification, identifies the mechanism to solve the problem among a set of known solutions (retry, majority voting, probing, error correction). Subsequently, DePer triggers the synthesis that enriches the previously synthesized CONNECTor with the suggested mechanism. The three main contributions are: (a) an enhancement

of the CONNECTor, taking into account performance and dependability mechanisms, (b) a more detailed CONNECTor adaptation process -related to the performance and dependability mechanisms, and (c) an implementation of the stochastic model-based analysis. We first investigated a scenario considering black box NSs and then, by relaxing this assumption, we considered another scenario where NSs trust and authorize CONNECT to access them in order to apply some mechanisms that enhance the synthesized CONNECTor.

2.2.4 Experiments

In order to validate the CONNECT solution to on-the-fly mediator synthesis, a significant part of our effort during the final year has been devoted to software prototype implementation together with integration within the overall CONNECT architecture and experimentation using real world case studies. Hence, much of our effort has been done collaboratively with WP1 and WP6 and the interested reader is referred to Deliverables D1.4 and D6.4 for detail about integration, resp. experiment work.

Still, we would like to highlight here the following experiments with automated mediator synthesis, ranging from dealing with application-layer heterogeneity only, to cross-layer application and middleware heterogeneity:

- *Universal Instant Messaging:* This work considers enabling universal instant messaging through the automated synthesis of mediators bridging existing instant messaging applications over TCP. Hence, universal instant messaging is focused on application-layer interoperability only. In addition, a noticeable feature of the target application is that interoperability relies on 1-1 message mappings as opposed to more complex N-M mappings. This results in a simpler solution to CONNECTor synthesis where the computation of mappings and that of mediator synthesis are realized in tandem using the OFSP ontology-based model checking introduced in Deliverable D3.2. The overall solution to universal instant messaging is detailed in Appendix D on "*Achieving Interoperability through Semantics-based Technologies: The Instant Messaging Case*", which is a paper published at ISWC'12.

- *Interoperable File Management:* Interoperable file management has been taken for experimenting with our solution to mapping-based mediator synthesis that is detailed in in Appendix A, ignoring middleware heterogeneity. Hence, this use case illustrates application-layer only interoperability as we focus on interactions over HTTP. However, the application is representative of the case where 1-N mappings need to be implemented. We specifically consider enabling access to files stored using the various Web-based file management systems that are now available (e.g., iCloud, GoogleDrive, MS Skydrive), independent of the specific client application that is deployed on the user's terminal. As detailed in Appendix A, results show that the overhead of mediation in the interaction with Web-based file management systems, is reasonable.

- *Interoperable Conference Management:* Compared to the interoperable file management discussed above, the case of interoperable conference management introduces the requirement for cross-layer interoperability as target systems run over heterogeneous middleware and middleware-layer messages embed application-specific information. Specifically, we consider the case of interoperability between an Amiando[6] client (resp. service) with a Regonline[7] service (resp. client). Both Amiando and Regonline are based on the request/response paradigm, i.e., the client issues a request that includes the appropriate parameters and the server returns the corresponding response. However, Amiando is developed according to the REST architectural style, uses HTTP as the underlying communication protocol, and relies on JSON for data formatting. On the other hand, Regonline is implemented using SOAP, which implies using WSDL to describe the application interface, and is further bound to the HTTP protocol. Our solution to cross-layer protocol mediation is sketched in Section 2.2.2 and detailed in Appendix B. In particular, experiment with Amiando et Regonline shows that the proposed cross-layer mediation introduces an acceptable overhead compared to interactions using dedicated custom client applications.

[6] http://developers.amiando.com
[7] http://developers.regonline.com

- *Mediators for GMES:* Experiment using the GMES use case is the focus of the CONNECT Work package WP6 that addresses the integration of the various CONNECT enablers to deal with on-the-fly connection of the various heterogeneous devices involved in GMES scenarios. This has in particular allowed us to experiment with the CONNECT synthesis enabler in the case of connection of NSs using middleware based on heterogeneous interaction paradigms, namely publish-subscribe and client-service.

3 CONNECTor Synthesis: From Theory to Practice

The core contribution of the WP3 work is to enable interoperability between highly heterogeneous systems by reconciling behavioral discrepancies, or mismatches, that occur between functionally compatible systems, from the application down to the middleware layer. The contribution is both theoretical and practical, decomposing into:

1. *A theory of mediators* that formalizes the process of mediator synthesis, and in particular highlights the central role of ontologies in reasoning about functional and behavioral matching of networked systems. Detail about the theory may be found in Deliverable D3.3 while Section 3.1 briefly recalls its main features.

2. *A synthesis enabler* that is an implementation of the proposed theory, further targeting fully automated mediator synthesis, which can then be performed on the fly following the run-time discovery of NSs that implement functionally matching affordances. As discussed in the previous chapter, most of our effort during the reporting period has been focused on finalizing the prototype implementation of the synthesis enabler and integrating it in the overall CONNECT architecture. Section 3.2 outlines the main features of the synthesis enabler regarding automated support for computing interface mappings and synthesizing associated mediators.

While the synthesis enabler is a core constituent of the CONNECT architecture to enable emergent middleware, it may be used more generally to develop and experiment with interoperability solutions at design- as well as run-time. This in particular includes being able to compare the various approaches to mediator synthesis that are being developed by the research community. Toward this goal, we have started the development of an extensible toolbox for mediator synthesis, which is introduced in Section 3.3.

3.1 The CONNECT Theory of Mediators

Figure 3.1 depicts the mediator synthesis process given NSs models and the ontologies describing the domain-specific knowledge. We further recall that the CONNECT mediator synthesis is preceded by *Functional Matching* and *Middleware Abstraction*: (i) the former consists in checking whether, at a high level of abstraction, the functionality (as specified by the affordance) required by one system can be provided by the other; (ii) the latter abstracts/translates (sequences of) middleware functions into the input/output qualification of actions. Then, the mediator synthesis process is made up of three phases or steps:

1. *Identification of the Common Language* makes comparable the NS behaviors by determining their common language and, possibly, reduces their size and speeds up the reasoning about them (see Figure3.1 ①).

2. *Behavioral Matching* (see Figure3.1 ②) checks the NS compatibility, possibly identifying behavioral mismatches.

3. *Mediator Synthesis* (see Figure3.1 ③) produces a mediator that address the identified mismatches between the two NSs and allows them to communicate.

We outline the principles of the aforementioned steps in what follows, after recalling the definition of the ontology-based NS model that they use.

3.1.1 Ontology-based Networked System Model

The NS *interface* defines the set of observable *actions* that the NS requires/provides from its running environment. The NS *behavior* further describes the interactions of the NS with its environment, and defines how the actions of the NS interface are used. We formalize the Behavior through enhanced Labeled Transition Systems (eLTS) that are Labeled Transition Systems [66] enhanced with explicit final states and whose labels are structured to explicitly model input/output actions and data. More formally:

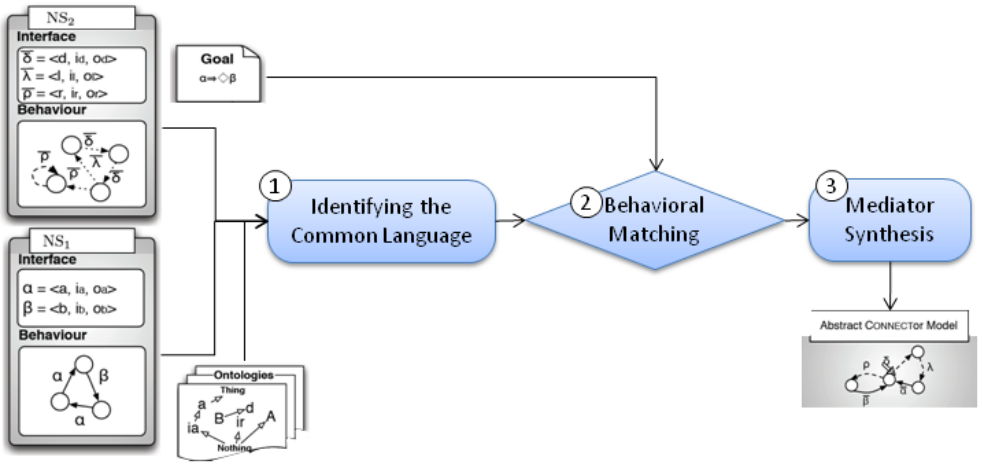

Figure 3.1: The Mediator Synthesis process

Definition 1 (*enhanced Labeled Transition Systems*) *An enhanced Labeled Transition Systems (eLTS) P is a quintuple (S, L, D, F, s_0) where:*

- *S is a finite non-empty set of states;*
- *L is a finite set of labels describing ontologically-annotated actions with data (i.e., input and output parameters); L is called the alphabet of P:*
- *$D \subseteq S \times L \times S$ is a transition relation;*
- *$F \subseteq S$ is the set of final states;*
- *$s_0 \in S$ is the initial state;*

We use the usual following notation to specify transitions: $s_i \xrightarrow{l} s_j \Leftrightarrow (s_i, l, s_j) \in D$, which denotes that P transits from s_i to s_j after performing l.

eLTSs can then be combined using the parallel composition operator, where we adopt the semantics of CCS, and consider synchronization over compatible actions:

Definition 2 (*Compatible Actions*) *Let $<a_1, In_1, Out_1>$ and $<a_2, In_2, Out_2>$ be two complementary ontologically-annotated actions where $a_1 = act$ and $a_2 = \overline{act}$. We say that a_1 and a_2 are compatible actions, denoted by $a_1 =_C a_2$ iff $(\forall i_j \in In_2, \exists i_i \in In_1 \mid i_j \sqsupseteq i_i) \wedge (\forall o_i \in Out_1 \; \exists o_j \in Out_2 \mid o_i \sqsupseteq o_j)$ where $\sqsubseteq$ denotes the ontological subsumption.*

The initial state together with the final states, define the boundaries of the protocol's coordination policies or traces. A *coordination policy* or *trace* is indeed defined as any sequence of actions that starts from the initial state and ends into a final state. Then, a coordination policy or trace represents a communication (i.e., coordination or synchronisation) unit and is formally defined as follows:

Definition 3 (*Trace* or *Coordination Policy*) *Let $P = (S, L, D, F, s_0)$ be an eLTS. A trace $t = l_1 \; l_2 \; \ldots \; l_n \in L^*$ is such that: $\exists (s_0 \xrightarrow{l_1} s_1 \xrightarrow{l_2} s_2 \; \ldots \; s_m \xrightarrow{l_n} s_n)$ where $\{s_1, s_2, \ldots, s_m, s_n\} \in S \wedge s_n \in F$.*

We use the usual compact notation $s_0 \stackrel{t}{\Rightarrow} s_n$ to denote a trace, where t is the concatenation of actions of the trace.

3.1.2 The Theory of Mediator in a Nutshell

This section summarizes the different steps of the mediator synthesis process given the protocols (i.e., eLTSs) of two NSs to CONNECT.

Identification of the Common Language

If the protocols are behaviorally compatible, this implies that at a given level of abstraction, there exists a common language to which a subset of the concepts of the ontologies of P and Q can be mapped to according to the subsumption relation. Still, NSs may interact with third parties using actions that do not belong to the common language.

Computation of the common language is implementation specific; we use constraint-based programming in the CONNECT synthesis enabler.

Behavioral Matching

Given the eLTSs of the NSs, which are relabeled using the common language, the behavioral matching step verifies the behavioral compatibility of NSs by seeking compatible traces modulo: ordering mismatch, third party interactions and extra actions. A successful behavioral matching implies that a mediator exists and it will be automatically synthesized at the subsequent step.

The output of the matching includes: (i) the behavioral compatibility relation (i.e., non-matching, intersection, inclusion, and total matching); (ii) the compatible traces –identifying sub-eLTSs – labeled by actions and data concepts of the domain ontology including silent actions τs for third party actions; (iii) an eLTS defined over the domain-specific ontology including τ that realizes a skeleton that then needs to be refined to become a mediator.

Mediator Synthesis

Given two behaviorally compatible protocols P and Q, we want to synthesize a mediator M such that the parallel composition $P||M||Q||E$ where E is the environment, allows P and Q to evolve to their final states. Actions of P and Q can belong either to the *common language* or the *third party language*, i.e., the environment language. We build the mediator in such a way that it lets P and Q evolve independently for the portion of the behavior to be exchanged with the environment (denoted by the τ action) until they reach a "synchronization state" from which they can synchronize over compatible actions. We recall that the synchronization cannot be direct since the mediator needs to perform suitable manipulations as for instance actions reordering or translation used to identify the common language.

According to the above, the mediator M is composed of two separate components: M_C and M_T. M_C speaks only the common language and M_T speaks only the third parties language. M_C is obtained by merging the eLTSs identified for pairs of compatible traces. The final mediator is obtained by translating each action or sequence of actions of the common language into the corresponding concrete (sequence of) action(s) of the languages of P and Q. M_T, if it exists, is built starting from the third party language of P and Q and specifies synchronization with the environment.

3.2 The CONNECT Synthesis Enabler

This section outlines the specific implementation of the mediator theory by the CONNECT synthesis enabler, which addresses the "identification of the common language" as an interface mapping problem.

3.2.1 Mapping Interfaces

A sequence of actions:

$$\langle \alpha_i = \langle a_i, I_{a_i}, O_{a_i} \rangle_{i=1..m} \in \mathcal{I}_1 \rangle$$

required by an NS can be achieved using a sequence of actions:

$$\left\langle \overline{\beta_i} = \langle \overline{b_i}, I_{b_i}, O_{b_i} \rangle_{j=1..n} \in \mathcal{I}_2 \right\rangle$$

provided by the other NS only if some constraints are verified.

The first constraint is that the provided operations are a specialization of the required ones, that is:

$$\bigsqcup_{j=1}^{n} b_j \sqsubseteq \bigsqcup_{i=1}^{m} a_i.$$

When an NS requires an action, it sends the input data and receives the corresponding output data. Hence, we can allow the NS to progress if the input data are cached and it does not expect any output data. Let us set l as the position of the first required action that necessitates some output data, i.e.:

$$\forall h \in [1, l[,\ O_{a_h} = \emptyset.$$

To provide this output data, the corresponding provided actions have to be executed. Therefore, the input data of the first action can be obtained using the input data already received, i.e.:

$$\bigsqcup_{i=1}^{l} I_{a_i} \sqsubseteq I_{b_1}.$$

Once this action has been executed, then the cached data are augmented with its output data. In general, when $i-1$ provided actions have been performed, the cache is:

$$cache_{i-1} = \left(\bigsqcup_{j=1}^{l} I_{a_j}\right) \bigsqcup \left(\bigsqcup_{h=1}^{i-1} O_{b_h}\right).$$

Then, the input of the i^{th} action should be obtained using the cached data, i.e.,

$$cache_{i-1} \sqsubseteq I_{b_i}.$$

Once all the provided actions have been performed, we can execute the remaining required actions if all their output are available. i.e.:

$$\forall h \in [l, m], cache'_h \sqsubseteq O_{a_h}$$

where the cache is augmented with the input data of the required actions, that is:

$$cache'_h = \left(\bigsqcup_{i=l}^{h} I_{a_i}\right) \bigsqcup cache_n.$$

We use constraint programming to compute the mapping efficiently, where the above conditions on the operations and input/output data represent the constraints. The solver implements intelligent search algorithms such as backtracking and branch and bound to optimize the time for finding the mapping.

Figure 3.2 illustrates a many-to-many mapping process, written $Map(\langle \alpha_1, ..., \alpha_l, ..., \alpha_m \rangle, \langle \overline{\beta_1}, ..., \overline{\beta_n} \rangle)$. The $l-1$ first required operations are performed since they do not require any output. To execute β_1, we need to transform the cached input data into the input expected by β_1 using the translation function f_1. In general, input data translation is performed by f_i functions while the output data are translated by the g_j functions until the mapping is completed.

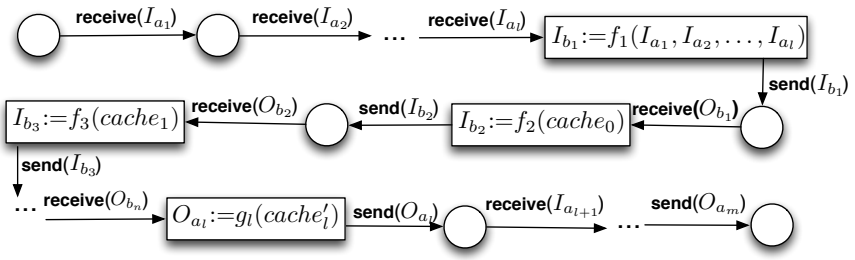

Figure 3.2: Many-to-many Mapping

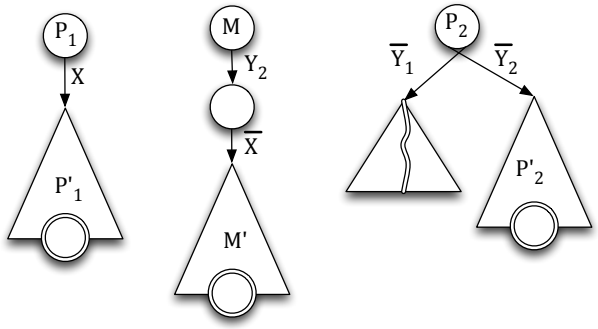

Figure 3.3: Incremental Synthesis of a Mediator

3.2.2 Synthesizing Abstract Mediators

While the computation of interface mapping ensures that a sequence of actions from one NS can safely be achieved using another sequence of actions from the other NS, it does not specify when each mapping has to be performed. Furthermore, the interface mapping might be ambiguous. Hence, the mappings need to be combined such that the interaction of the two NSs does not lead to erroneous states, e.g., a deadlock. We analyze the behavioral specifications of the two NSs, which are specified using eLTSs (say P_1 and P_2, respectively) and generate a third eLTS, the mediator M, such that the mediated NSs reach their final states, which implies that the system made up of the parallel composition of P_1, P_2, and M is free from deadlocks, or we determine that no such mediator exists. If a mediator exists, then we say that P_1 and P_2 can interoperate using a mediator M, written $P_1 \leftrightarrow_M P_2$.

We inductively build a mediator M by forcing the two NSs to progress consistently so that if one requires the sequence of actions X, the interacting NS must provide a semantically-compatible sequence of actions $\overline{Y}$. Given that an interface mapping guarantees the semantic compatibility between the actions of the two NSs, the mediator is able to compensate for the differences between their actions by performing the necessary transformations. The base case is that where both P_1 and P_2 are final states, in which case the mediator is made up of one final state. Then, at each state we choose an eligible mapping, i.e., both processes can engage in the sequences specified by this mapping and moves to states where a (sub) mediator can be generated, which is formally described as follows:

if	$P_i \stackrel{X}{\Rightarrow} P_i'$ and $\exists(X, \overline{Y}) \in Map\,(\mathcal{I}_i, \mathcal{I}_{3-i})$
such that	$P_{3-i} \stackrel{\overline{Y}}{\Rightarrow} P_{3-i}'$ and $P_i' \leftrightarrow_{M'} P_{3-i}'$
then	$P_i \leftrightarrow_M P_{3-i}$ where $M = Map(X, \overline{Y}); M'$

This implies that when multiple mappings can be applied, we select one of them and check if it allows the two processes (P_i and P_{3-i}) to reach their final states. Otherwise, we backtrack and select another mapping.

Consider the example in Figure 3.3. P_1 is ready to engage in X, P_2 can execute either $\overline{Y_1}$ or $\overline{Y_2}$, and X can be mapped to both $\overline{Y_1}$ and $\overline{Y_2}$. If we select the first mapping, the final state of P_2 cannot be reached. Hence, we backtrack and select the second one. To compute the mediator M, we append the mapping of $Map(X, \overline{Y_2})$ to M', which is the mediator computed between P_1' and P_2'. If none of the mappings is applicable, then we are not able to generate the mediator. Note however, that the mediator is not unique since many mappings may lead P_i and P_{3-i} to their final states. In this case, we only keep the first valid mapping.

As outlined in the previous chapter, the CONNECT synthesis enabler is available for download from the CONNECT Web site and has been integrated within the overall CONNECT architecture to support the realization of emergent middleware.

3.3 Beyond CONNECT Architecture: Toward a Mediator Synthesis Service

We have developed our automated solution to mediator synthesis as part of the overall CONNECT architecture enabling emergent middleware. Our contribution over the state of the art primarily lies in handling interoperability from the application to the middleware layer in an integrated way, while related work addresses heterogeneity at one of the two layers, ignoring the other. In addition, our intent is not to advocate a unique solution for the automated synthesis of mediators. Rather, we have introduced a general theory for mediator synthesis and experienced it with different implementation variants such as:

- Synthesis using ontology-based model checking that deals with 1-to-1 operation mappings and that is implemented using the OLTSA tool (see Appendix D);
- Mapping-based synthesis with the associated constraint-based generation of interface mappings that supports N-to-M operation mappings as well as ambiguous mappings (see Appendix A).

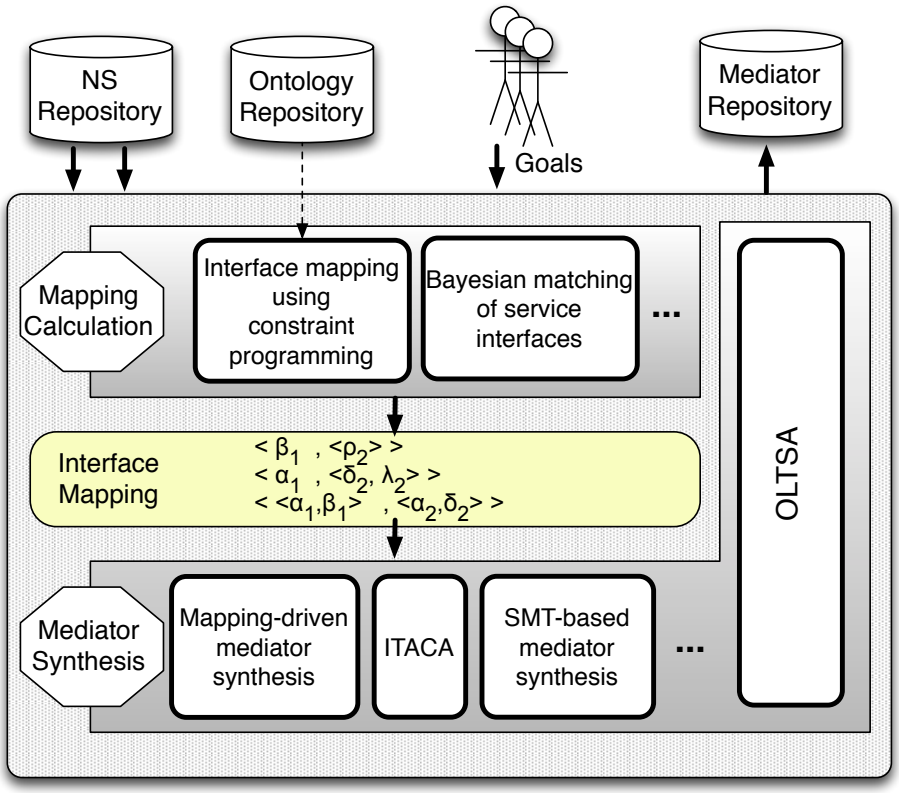

Figure 3.4: Mediator Synthesis Service

Beyond the above CONNECT solutions, existing work in the area of mediator synthesis brings relevant synthesis algorithms that may be considered as adequate alternatives, depending on the specifics of the networked systems that need be connected. In general, it may prove useful for the community to provide a "Mediator synthesis service" that allows plugging in algorithms for mediator synthesis. In addition to enabling selecting the synthesis approach that suits best the given context, such a service would allow comparing existing approaches from both a qualitative and a quantitative perspective.

Toward the above goal, we have been elaborating the high level architecture of a mediator synthesis service, which is depicted in Figure 3.4. The proposed architecture accounts for the diversity of existing approaches to mediator synthesis where: some require the use of ontologies and other not; some require interface mapping (also called adaptation contracts) and other not; and some make use of goals and

other not. For example, to compute the interface mapping, we may rely on a bayesian approach such as that devised by Kimelman *et al.* [67]), which matches interfaces based on various features (e.g., labels similarity, schema structure, etc.). Similarly, we can make use of different algorithms for the synthesis of mediators based on the generated interface mapping, e.g., ITACA [32] and SMT-based synthesis [16].

We are currently refining the architecture of the ”Mediator synthesis service”, which we intend to release shortly as a Web service for use by the community.

4 Assessment

The CONNECT Description of Work sets a number of assessment criteria for our approach to automated CONNECTor synthesis. In brief, the criteria relate to evaluating the efficacy of our approach regarding the mismatches that may actually be solved and its ability to be fully automated. The proposed assessment criteria for the CONNECT approach to the runtime synthesis of CONNECTors were further classified according to the following target objectives in Deliverable D6.3:

O1. Overcoming a large range of protocol mismatches in an automated way;

O2. Performing CONNECTor synthesis on the fly and at runtime;

O3. Devising compositional approaches to the synthesis of CONNECTors;

O4. Accounting for QoS aspects of the NSs interaction.

Use case	**Application-layer Heterogeneity**	**Middleware-layer Heterogeneity**
Universal Instant Messaging (see Appendix D)	1-1 operation mismatches	–
Interoperable File Management (see Appendix A)	1-N operation mismatches	–
Interoperable Conference Management (see Appendix B)	1-N operation mismatches	Heterogeneous client-service middleware
GMES (see Deliverable D6.4)	1-N operation mismatches	Heterogeneous coordination paradigms

Table 4.1: Assessing the CONNECT Synthesis Enabler with Real-world Scenarios

Building on the extensive experiments carried out in the final year of the CONNECT project, this chapter provides an assessment of the CONNECT synthesis enabler against each of the above objectives. As outlined in Section 2.2.4, the undertaken experiments relate to the GMES use case but also a number of Internet-based services. Table 4.1 classifies the types of heterogeneity that arise with the case studies that we have been considering. In brief, application-layer behavioral mismatches range from 1-1 to 1-N, while we did not encounter the N-M case in any of the scenarios although this is well supported by the synthesis enabler. As for middleware-layer behavioral mismatches, these were encountered in two case studies, including the presence of heterogeneous coordination paradigms. In addition to the real world use cases given in Table 4.1, we have also assessed the CONNECT synthesis enabler against the SWS challenge that introduces the Purchase Order Mediation scenario [98]. The scenario provides common ground to discuss semantic (and other) Web Service solutions and compare them according to the set of features that a mediation approach should support. However, this scenario does not introduce any additional heterogeneity dimension compared to the Interoperable file management use case. We thus do not detail it here and refer the interested reader to Appendices A and C.

O1. Overcoming Protocol Mismatches in an Automated Way

We need to assess the synthesis enabler against the common protocol mismatches that it indeed enables us to overcome. As discussed in previous deliverables, common protocol mismatches decompose into:

(i) Signature mismatch,

(ii) Splitting of actions,

(iii) Merging of actions,

(iv) Extra send or missing receive,

(v) Extra receive or missing send,

(vi) Ordering mismatch.

While Mismatch (v) cannot be addressed in general since it requires the auto-generation of actions, all the other mismatches are addressed by the synthesis enabler. However, Mismatches (iii) and (vi) are addressed under the synchronous semantics. Practically, this complies with the requirements of the use cases of Table 4.1.

In addition, the synthesis enabler supports the combination of the above mismatches (but (v)), still under the synchronous semantics, as it handles N-M and ambiguous mappings although these were not required to support the use cases that we studied.

O2. Performing On-the-fly CONNECTor Synthesis

In addition to assessing the ability of the synthesis enabler to overcome basic protocol mismatches, one needs to assess its performance in doing so, especially regarding the offered response time. The following analyzes the performance of the synthesis enabler, examining each of the use cases of Table 4.1 in turn.

Universal Instant Messaging Use Case: Consider first the universal instant messaging use case that is detailed in Appendix D and for which we have introduced a dedicated ontology. We have then defined the CONNECT-compliant models of MSNP-, YMSG-, and XMPP-based systems, introducing their semantically annotated interfaces and behavioral specifications. Given that this specific use case allows for 1-to-1 interface mapping, we use OLTSA to jointly perform the necessary behavioral matching and synthesize the corresponding mediator model. Starlink is further exploited for the parsing and composing of messages. Figures 4.1 provides the performance of message exchange between two Instant messaging NSs for a message size of 100 characters, considering both homogeneous and heterogeneous NSs. For all configurations, we consider both native interactions -if eligible- and interactions using a CONNECTor. In particular, besides native interactions between homogeneous NSs, we have native interactions between MSNP and YMSG through the proprietary mediator that is embedded within MSNP and YMSG. The overhead of the CONNECTor in the case of interactions between homogeneous protocols depends on the types of the instant messaging protocols: while the overhead is negligible for the XML-based XMPP system, it is significant in the case of the binary YMSG protocol. Still, considering the response time experienced by the end-user, the overhead of CONNECTor execution is acceptable since the largest experiences response time remains close to that of native XMPP communication. It is further worth noticing the performance of MSNP/YMSG interoperability using a CONNECTor that introduces an overhead of 40% compared to the optimized, proprietary interoperability solution. This is overall a good result as the OLTSA-based mediator synthesis is an early prototype that does not embed advanced optimization for CONNECTors.

Interoperable File Management Use Case: We now concentrate on the performance of mediator synthesis using the mapping-based synthesis enabler. We more specifically explore interoperable file management where a WebDAV client (Mac Finder) accesses the Google Docs service seamlessly using a CONNECTor. This specific use case is detailed in Appendix A and is only briefly outlined here. The CONNECTor synthesis relies on the NEPOMUK File Ontology[1] (NFO), which we extended with the operation-related concepts, to describe the file management domain. We then described the interfaces of the two applications, and annotate them using the specified ontology. Finally, we defined the corresponding eLTSs according to documentation of the protocols and our understanding thereof. We used the synthesis enabler to generate the mediator and deploy it on top of an Apache Tomcat server with the Milton API[2] to parse WebDAV messages. We measured the time to perform a simple interaction scenario, which consists in authenticating, moving a file from one folder to another, and listing the content of the two folders. This leads to illustrate 1-to-N operation mapping by the CONNECTor as the move operation of WebDAV translates into three Google Docs operations, i.e., download, upload and delete. As for performance measurements, the file is a 4KB text document to lessen the network delay. Figure 4.2 provides the resulting response time, not counting the authentication phase. We can see that the mediator introduces

[1] http://www.semanticdesktop.org/ontologies/nfo/

[2] milton.ettrema.com/

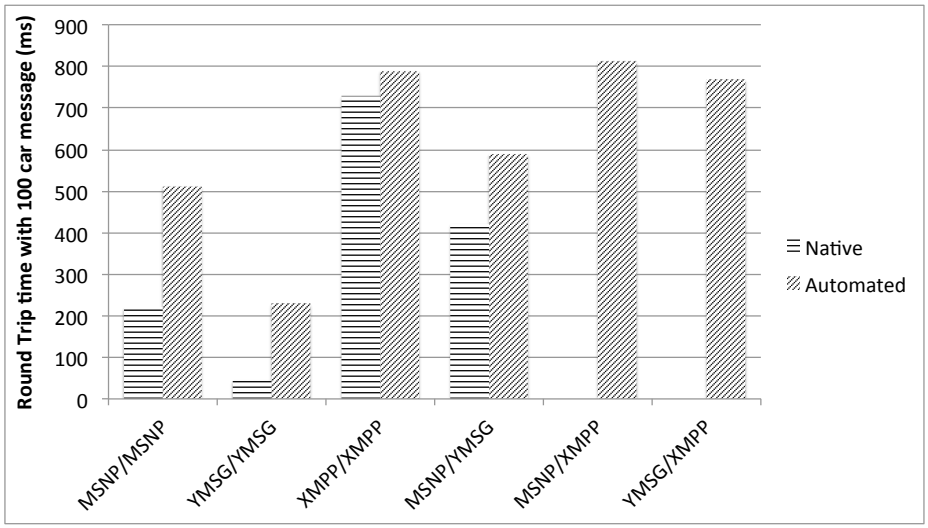

Figure 4.1: Mediated Instant Messaging

only 17% time overhead compared to WebDAV-only interaction while it keeps performance close to that of GDoc-only interactions.

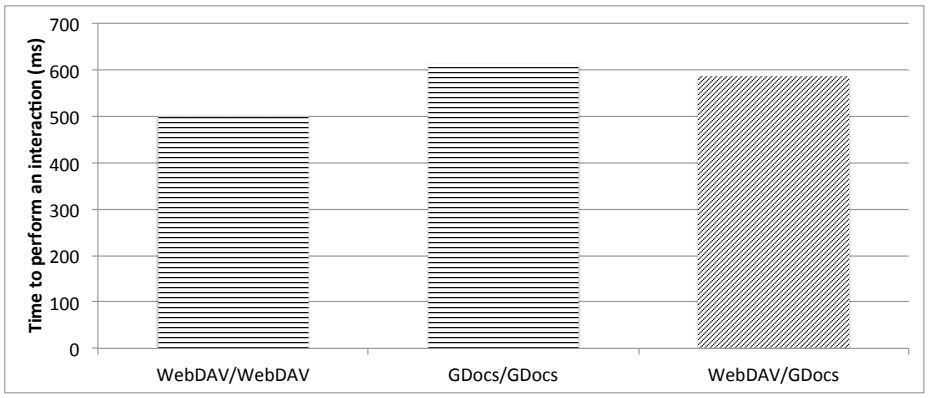

Figure 4.2: Mediated File Management

Interoperable Conference Management Use Case: The next use case is concerned with the interoperable conference management case study that is detailed in Appendix B and requires dealing with cross-layer protocol mediation. Indeed, we focus on interoperability between the RegOnline and Amiando conference management systems and compare results with the baseline, non-mediated, systems. On the server-side, we use the services provided by Amiando and Regonline. On the client-side, we use a Java implementation provided by Amiando, while for Regonline, we generate the client using the `wsimport`[3] tool and the WSDL service description. We compare the mediated execution-time with the non-mediated case. Each test was repeated 30 times, in similar conditions, and connection delays were excluded (e.g., opening sockets, SSL handshake, etc). Figure 4.3 depicts the execution-time overhead of the mediation. Since this test is performed using the real online services, the response time varies depending on the network conditions. As expected, the mediated execution-time is superior to the non-mediated case, given that the number of messages exchanged is doubled. We show the decomposition of the execution-time for mediation, composing and access/parsing. Network access and parsing cannot be distinguished in this case because parsing is done in multiple steps when data is available on the communication channel. While

[3]http://docs.oracle.com/javase/6/docs/technotes/tools/share/wsimport.html

the overhead of mediation and message composition is low, we see that parsing and network reception introduce the largest overhead. Further analysis that is detailed in Appendix B shows that the Amiando/RegOnline mediator execution-time can be reduced by using a more efficient SOAP Atomic parser, hence suggesting that enhanced response time may be obtained for CONNECTors through some engineering effort. Still, compared to the non-mediated tests, we can conclude that our mediation approach introduces an acceptable overhead while enabling seamless interoperability between the two systems, although the synthesis enabler prototype has not yet undergone extensive optimizations.

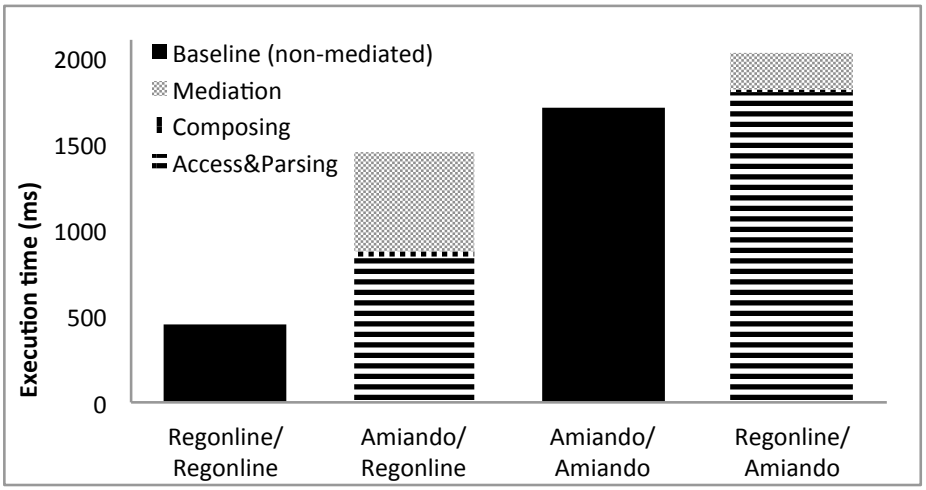

Figure 4.3: Mediated Conference Management

GMES Use Case: We conclude the analysis with the GMES use case that further exemplifies interoperability between NSs relying on middleware implementing heterogeneous coordination paradigms. We specifically compare the time taken for mediated and non-mediated conversations for different services (see Figure 4.4): weather, positioning, and vehicle control. In Country 1, the Command and Control Center (C2) uses SOAP to interact with SOAP-based services: weather station, positioning-A, and UGV (Unmanned Ground Vehicle). In Country 2, the weather service and the UAV (Unmanned Aerial Vehicle) are SOAP-based services for which we generate appropriate clients using WSDL2Java Axis command[4], while the positioning-B service is an AMQP publisher for which we build an appropriate subscriber using the RabbitMQ library[5]. In order for C2 to use the services provided by Country 2, mediators have to be created, which reconcile the behaviors and data of C2 clients with Country 2 NSs. In our experiment, the synthesized mediators are deployed using Starlink (see Deliverable D1.4).

For each service, we measured the average time necessary to perform a meaningful conversation with each system of each country as well as between C2 and services from Country 2. In the case of the weather service, the conversation includes authentication, obtaining weather information using a single `getWeather` operation in Country 1 and two operations, `getTemperature` and `getHumidity` in Country 2, and then logging out. In the case of the positioning service, a unique operation `getPosition` is performed but using different middleware paradigms: client/service and publish/subscribe. In the latter situation, the mediator has access to the data already in the queue and published by the Positioning-B service. In the vehicle control scenario, conversations consist in authentication, takeoff (in the case of UAV only), moving and turning both left and right, landing (in the case of UAV only), and logging out.

We deployed all the systems on a single Mac computer with a 2,7 GHz processor and 8 GB of memory to avoid network delays. We repeated each conversation 50 times and computed the average duration. The results are presented in Figure 4.5. We can see that in the case of positioning, the overhead is almost inexistent since there is no processing time on the server as the data are already published. In the case of the weather service, the mediated conversation takes twice the time required to interact with

[4] http://axis.apache.org/axis2/java/core/docs/userguide-creatingclients.html
[5] http://www.rabbitmq.com

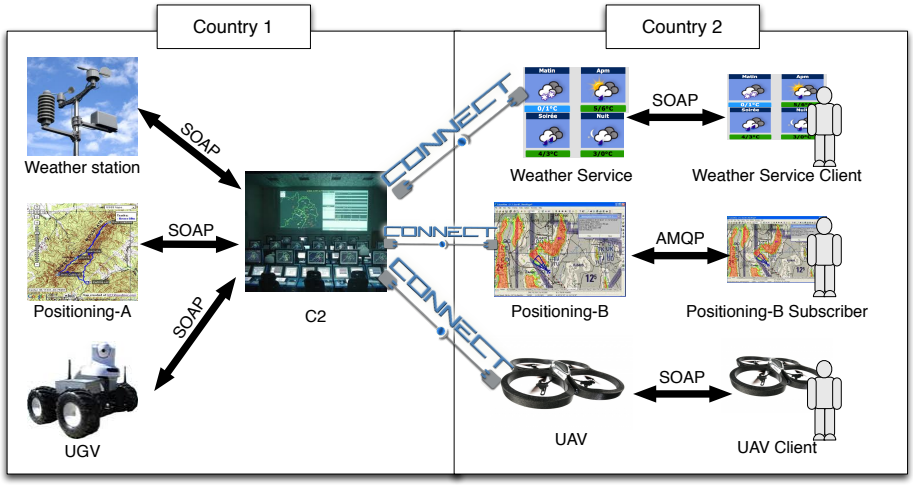

Figure 4.4: Evaluation Configuration for the GMES Use case

the service using the built-in client. This is due to the use of Starlink underneath. While Starlink provides a very flexible and generic approach to the generation of parsers and composers, it comes with a considerable performance cost mediation. This is one of the reasons that led us to investigate CCL parsers and composers detailed in Section 2.2.2, which, even though less generic than Starlink, provide better performance by using specific, manually created and optimized libraries (see Interoperable conference management above). This problem is exacerbated in the case of vehicle control where mediated conversations need 2.6 times more time to be performed. The reason is the way extra actions (`takeoff` and `land` in our use case) need to be represented in the k-colored automata used by Starlink. During the concretization step, i.e., from eLTS to k-colored automata of the mediator, we have to introduce extra states with `noaction` transitions to specify extra provided operations.

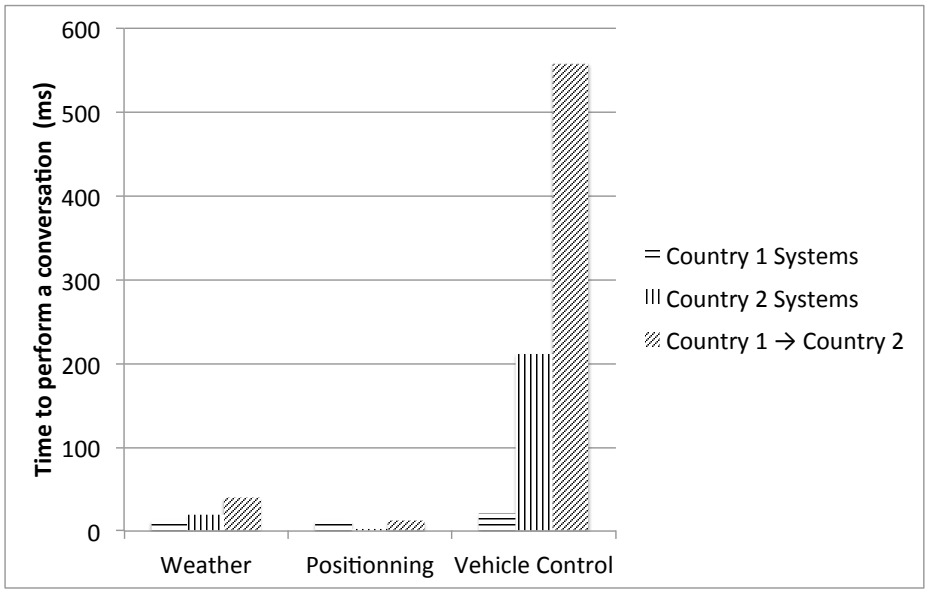

Figure 4.5: Time for Interaction between GMES Systems

A complementary consideration in assessing the ability of the synthesis enabler to perform on the fly connector synthesis is the ability of the enabler to handle partial models/observations of the interaction behaviors of the considered NSs. This indeed holds as long as the partial model is an abstraction of the complete model, which is given by construction of the CONNECT learning and synthesis enablers.

O3. Devising Compositional Approaches to the Synthesis of CONNECTors

Synthesized CONNECTors are subject to changes with respect to both functional and non-functional concerns. Reasons are numerous and relate to the changing environment as well as to the evolving knowledge about the CONNECTed systems thanks to the monitoring enabler.

By construction, CONNECTors cannot be built by composing previously synthesized CONNECTors, except for the synthesis of parsers and composers. However, not all steps of the synthesis process need to be applied when a CONNECTor has to evolve. Indeed, we recall that a CONNECTor is synthesized according to the following steps:

1. Synthesis of the parsers and composers that are necessary to interact with each of the NSs to be CONNECTed and that may be built though the composition of atomic parsers and composers (see Deliverable D1.4 and Appendix B).

2. Acquisition of the models of the NSs to be made interoperable, including the behavior associated with the target affordance for the CONNECTion and possible specific goal (see Deliverables D1.4 and D4.4).

3. Computation of the semantic mappings between the operations of the two NSs, given the NS's models and related domain ontology (See Appendix A).

4. Synthesis of the mediator by scheduling the operation mappings in a way that allows the two NSs to successfully coordinate, possibly driven by a given goal (See Section 2.2.1 and Appendix A).

Then, as discussed in Section 2.2.3, any change that impacts any of the above steps leads to re-run the synthesis process from the first step that is impacted. This reflects also what have been investigated by WP2 about compositionality of modular connectors (see Deliverable D2.4) where it is stated that compositionality holds for specific kind of changes while, in the worst case, repeating the entire synthesis process cannot be avoided.

O4. Accounting for QoS Aspects of the NSs Interaction

Non-functional aspects have been extensively addressed in WP5, including the assessment and possible adaptation of CONNECTors with respect to required quality of service, which has been discussed in Section 2.2.3. The interested reader is further referred to Deliverable D5.4 for detail about the treatment of CONNECTability within CONNECT, and related assessment.

In addition, as outlined in Section 2.2.3 and detailed in Appendix C, we have investigated as part of work package WP3 the assessment of performance of CONNECTors during synthesis. We identified three general strategies that can be applied to enhance the performance of a CONNECTed system:

(i) Alternative CONNECTor behaviors slicing, which can be applied if at least one of the NSs has alternative protocol behaviors. In this case, the CONNECTor behavior is sliced and it mediates only a subset of the NSs alternatives;

(ii) tuning the upper bound number of loop iterations; several bounds are considered in the analysis and only the ones that help in satisfying the performance requirements are considered in the final synthesized CONNECTor.

(iii) Deployment configuration that highlights the most convenient deployment among three possibilities: all remote systems where the mediator and the NSs are deployed on separate machines, and local to a NS1 or local to a NS2 where the mediator is deployed on the same machine where NS1 or NS2 is running, respectively.

Table C.7 illustrates the results of the analysis considering the *Purchase Order Mediation scenario with Performance*. This scenario that is detailed in Appendix C, is a modified version of the Purchase Order Mediation scenario [98] that includes also performance requirements over the to be connected system, i.e., networked systems and connector. In summary, acting on (i) we identified three versions of the mediator and hence of the CONNECTed system (Blue1, Blue2, Blue3 on the columns of the tables).

Acting on (ii) we identified six scenarios by imposing six different upper bounds (5, 10, 20, 30, 40 and 50 on the raws of the tables) on the number of items. Acting on (iii) we identify three possible mediator deployment scenarios (each table refers to one of them). By combining the above described scenarios, we obtained 54 final configurations. The results of the analysis of the computed configurations are all reported in Table C.7 where bold numbers refer to configurations satisfying the target performance requirement.

Mediator deployed *remotely*

	Blue3	*Blue2*	*Blue1*
max 50 items	0,604	0,753	0,537
max 40 items	0,732	0,907	0,650
max 30 items	0,929	1,139	0,823
max 20 items	1,271	1,531	1,123
max 10 items	2,013	2,336	1,769
max 5 items	2,650	2,958	2,399

Mediator deployed *locally to Blue*

	Blue3	*Blue2*	*Blue1*
max 50 items	0,846	0,788	0,891
max 40 items	1,029	0,958	1,078
max 30 items	1,312	1,221	1,365
max 20 items	1,811	1,683	1,859
max 10 items	2,922	2,710	2,916
max 5 items	3,958	3,584	3,940

Mediator deployed *locally to Moon*

	Blue3	*Blue2*	*Blue1*
max 50 items	1,997	4,292	1,302
max 40 items	2,377	4,926	1,566
max 30 items	2,934	5,780	1,966
max 20 items	3,832	6,393	2,639
max 10 items	5,526	**8,849**	4,017
max 5 items	6,611	**8,849**	5,276

Table 4.2: Average Throughput Analysis Results

5 Conclusion

The automated synthesis of mediators is one of the core functionalities of the CONNECT solution to eternal interoperability, as it generates the abstract protocols that need to be executed for two functionally compatible networked systems to successfully coordinate despite mismatches in their respective interfaces and/or behaviors. This has led the CONNECT consortium to contribute to the field of mediator synthesis from both a theoretical and a practical perspective:

(i) On the theoretical side, we have introduced a theory of mediators that formalizes the process of mediator synthesis, further stressing the central role of domain knowledge engineering in the target process;

(ii) On the practical side, we have introduced a concrete synthesis enabler, which generate abstract mediators in the form of enhanced LTS given ontologically-annotated models of NSs to CONNECT.

Overall, our contributions in the area of automated mediator synthesis are manyfold and we would like to stress:

1. The conjoint handling of application and middleware-layer heterogeneities;
2. The fully automated synthesis of mediators while the vast majority of solutions requires the user intervention to elicit the necessary semantic mappings between the operations of the NSs that get CONNECTed and need to be made interoperable.
3. The leveraging of ontology-based domain knowledge for the automated synthesis of mediators, which in particular allows us to compute the semantic mappings of NS operations; in particular, compared to related effort in the semantic Web service domain and especially the WSMO solution to protocol mediation that automatically computes operation mappings, ours ensures the correctness of mediators, which is not the case of WSMO mediators.
4. The extensive experimentation with real world case studies, including the study of interoperability with popular Internet-based services.

This document has more specifically outlined our contributions in the area of automated mediator synthesis regarding the work performed in the final year and during the overall project duration. In the final year of the project, our focus has been mostly practical, focusing on the finalization of the CONNECT synthesis enabler and further assessment using concrete case studies. Results show that the synthesis enabler handles base protocol mismatches and further generates mediators whose execution overhead is acceptable from the end-user perspective. Next to this effort, we have also started investigating the adaptation of CONNECTors to cope with changes related to either functional or non-functional concerns. The work is so far mostly theoretical while integration in the concrete CONNECT architecture is area for future work.

Appendix: Papers on CONNECTor Synthesis

This appendix collates the papers produced by the CONNECT consortium in the final year of the project on the topic of CONNECTor synthesis. Most of them are still under submission, while they together comprehensively overview the project contributions in the area of CONNECTor synthesis.

Specifically, the following set of papers is attached:

A **Automated Synthesis of Mediators to Support Component Interoperability** by *Amel Bennaceur and Valerie Issarny*, to be submitted for publication: This paper details the CONNECT enabler for CONNECTor synthesis, which lies in building upon ontology reasoning and constraint programming to infer mappings between the interfaces of the systems to be connected. These mappings guarantee semantic compatibility between the operations and data of the interfaces. Then, we analyze the behaviors of components in order to synthesize, if possible, a mediator that coordinates the resulting mappings so as to make the components interact properly. The approach is formally-grounded to ensure the correctness of the synthesized mediator. The paper also demonstrates the validity of the approach by implementing the MICS (Mediator synthesis to Connecting Components) prototype and experimenting it with various case studies.

B **An Automated Mediation Framework for Cross-Layer Protocol Interoperability** by *Amel Bennaceur* (Inria), *Emil Andriescu* (Ambientic & Inria), *Roberto Speicys Cardoso* (Ambientic) and *Valerie Issarny* (Inria), submitted for publication: While existing approaches to interoperability consider either application or middleware heterogeneity separately, we have been stressing in CONNECT that this does not suffice in real world scenarios: application and middleware boundaries are ill-defined and solutions to interoperability must consider them in conjunction. In this paper, we propose such a solution, which solves cross-layer interoperability by automatically generating parsers and composers that abstract physical message encapsulation layers into logical protocol layers, thus supporting application layer mediation. Further, whenever possible, the framework automatically synthesizes the appropriate protocol mediators between interacting components based on the reasoning about the components functional and behavioral semantics. To demonstrate the validity of our approach, we show how the framework solves cross-layer interoperability between existing conference management systems.

C **Synthesizing Connectors meeting Functional and Performance Concerns** by *Antinisca Di Marco, Paola Inverardi and Romina Spalazzese* (UNIVAQ), submitted for publication: This paper introduces an evolution of the CONNECT synthesis approach so as to take into account in the synthesis process, together with functional concerns, also performance aspects. By reasoning on systems' specification, the first step of the approach produces a mediator that satisfies the functional requirements. The second step, by considering specific strategies, acts on the produced mediator to satisfy also the performance ones.

D **Achieving Interoperability through Semantics-based Technologies: The Instant Messaging Case** by Amel Bennaceur *(Inria),* Valerie Issarny *(Inria),* Romina Spalazzese *(Univaq) and* Shashank Tyagi *(IT Banaras Hind University)* published in ISWC 2012 - 11th International Semantic Web Conference (2012): Instant messaging is a representative example of the current need for emergent CONNECTors, where various competing applications keep emerging. To enforce interoperability at runtime and in a non-intrusive manner, mediators are used to perform the necessary translations and coordination between the heterogeneous applications. Nevertheless, the design of mediators requires considerable knowledge about each application as well as a substantial development effort. In this paper, we present the application of the CONNECT approach based on ontology reasoning and model checking in order to generate correct-by-construction mediators automatically. Further, the specific case of Instant Messaging allows dealing with interoperability using 1-1 mapping of the operations and thereby synthesizing a mediator using ontology-based model checking as opposed to first computing interface mappings and then synthesizing a mediator. We demonstrate the feasibility of our approach through a prototype tool and show that it synthesizes mediators that achieve efficient interoperation of instant messaging applications.

A Automated Synthesis of Mediators to Support Component Interoperability

Amel Bennaceur (Inria), and Valérie Issarny (Inria)

Abstract Interoperability is a major concern for the software engineering field, given the increasing need to compose components dynamically and seamlessly. This dynamic composition is often hampered by differences in the interfaces and behaviours of independently-developed components. To address these differences without changing the components, mediators that systematically enforce interoperability between functionally-compatible components by mapping their interfaces and coordinating their behaviours are required. Existing approaches to mediator synthesis assume that an interface mapping is provided which specifies the correspondence between the operations and data of the components at hand. In this paper we present an approach based on ontology reasoning and constraint programming in order to infer mappings between components' interfaces. These mappings guarantee semantic compatibility between the operations and data of the interfaces. Then, we analyse the behaviours of components in order to synthesise, if possible, a mediator that coordinates the resulting mappings so as to make the components interact properly. Our approach is formally-grounded to ensure the correctness of the synthesised mediator. We demonstrate the validity of our approach by implementing the MICS (Mediator synthesIs to Connecting Components) prototype and experimenting it with various case studies.

A.1 Introduction

Interoperability is a fundamental challenge for software engineering [68]. Today's systems are increasingly developed by reusing and integrating existing components. However, even though these components should be able to integrate since, at some high level of abstraction, they require and provide compatible functionalities, the conflicting assumptions that each of them makes about its running environment hinder their successful interoperation. Indeed, components cannot readily be reused when inconsistent semantics are buried in their interfaces or behaviours [50].

There exists a wide range of approaches to enable independent components to interoperate [102, 61]. Solutions that require performing changes to the components are usually not appropriate since the components to be integrated may be built by third parties (e.g., COTS—Commercial Off-The-Shelf—components or legacy systems); no more appropriate are approaches that prune the behaviour leading to mismatches since they also restrict the components' functionality [5]. Therefore, many solutions that aggregate the disparate components in a non-intrusive way, i.e., without changing the internal implementation of these components, have been proposed [106, 32, 82, 104, 61, 83, 27, 52]. These solutions use intermediary middleware entities, called *mediators* (also called connector wrappers [106] or mediating adapters [32]), to connect heterogeneous components despite disparities in their data and/or interaction models by performing the necessary coordination and translations while keeping them loosely-coupled.

Nevertheless, creating mediators requires a substantial development effort and a thorough knowledge of the application-domain. Moreover, the increasing complexity of today's software components, sometimes referred to as Systems of Systems [79], makes it almost impossible to develop 'correct' mediators manually. Therefore, formal approaches are necessary to characterise components precisely and generate mediators automatically [119, 24].

First, automatically generating mediators helps developers to manage the bulk of the systems they need to integrate. Indeed, developers increasingly have to incorporate to their systems convenient services such as instant messaging or social interaction. Although, most of these systems provide similar functionalities, their interfaces are usually heterogeneous. For example, Google Talk and Facebook chat both have instant messaging capabilities but expose them using different interfaces. Similarly, Facebook and Google+ allow social interaction between their users using distinct interfaces. By providing an automated mediation solution, the developer no longer has to struggle with the inflexibility of point-to-point data mapping since the bridging code is generated automatically. The automated generation of mediators also enables seamless and spontaneous interaction between components in highly dynamic and extremely heterogeneous environments by computing, at runtime, a mediator that ensures their interoperation [9].

Existing approaches to the automatic generation of mediators assume the correspondence between the operations of the mediated components to be given in terms of an *interface mapping* or an adaptation contract [24]. Identifying such correspondence requires not only knowledge about the components but also knowledge about the domain itself.

Research on knowledge representation and artificial intelligence has now made it possible to model and automatically reason about domain information crisply, if not with the same nuanced interpretation that a developer might [101]. In particular, *Ontologies* build upon sound logical theory to provide a machine-interpretable means to reason automatically about the semantics of data based on the shared understanding of the domain [6]. They play a valuable role in software engineering by supporting the automated integration of knowledge among teams and project stakeholders [30]. Ontologies have also been widely used for the modelling of Semantic Web Services and to achieve efficient service discovery and composition [8]. In particular, OWL-S (Semantic Markup for Web Services) uses ontologies to model both the functionality of a Web service and the associated behaviour, i.e., the protocol according to which it interacts [81]. Besides ontology-based modelling, WSMO (Web Service Modelling Ontology) relies on ontologies to support runtime mediation based on pre-defined patterns but without ensuring that such mediation does not lead to an erroneous execution (e.g, deadlock) [36]. Hence, although ontologies have long been advocated as a key enabler in the context of service mediation, no principled approach has been proposed for the automated synthesis of mediators by systematically exploiting ontologies. We argue that interoperability should not be achieved by defining yet another ontology nor yet another middleware but rather by exploiting the knowledge encoded in existing domain-specific ontologies together with the behavioural description of components and using them to generate mediators automatically. These mediators bridge heterogeneous components by delivering information when it is needed, in the right format, with the original business context intact.

This paper focuses on functionally-compatible components, i.e., components that at some high level of abstraction require and provide semantically compatible functionalities, but are unable to interact successfully due to mismatching interfaces and behaviours. We propose an approach that combines ontology reasoning and constraint programming in order to generate a mapping between the interfaces of functionally-compatible components. Then, we use the generated mappings and examine the behaviours of both components to automatically synthesise a mediator that ensures their safe interaction. The mediator, if it exists, guarantees that the system is free from deadlocks and blocking sends. Specifically, our contributions are:

- *Efficient interface mapping using semantic reasoning and constraint programming:* We reason about the semantics of data and operations of each application and use a domain ontology to identify the semantic correspondences the interfaces of components, i.e., interface mapping. Interface mapping guarantees that operations from one component can safely be performed using operations from the other component.

- *Automated synthesis of mediators:* We explore the behaviours of the two components and generate the mediator that composes the computed mappings so as to force the components to coordinate synchronously. The mediator, if it exists, guarantees that the mediated system is deadlock-free.

- *Tool-support for automated mediation:* We further demonstrate the feasibility of our approach through the MICS (Mediator Synthesis to Connecting Components) tool and illustrate its usability using real-world scenarios involving heterogeneous components. Our semantic-based approach to the automated generation of mediators leads to a considerable increase in the quality of interoperability assurance between heterogeneous components: it significantly reduces the programming effort and ensures the correctness of the mediation while preserving efficient execution time.

This paper is organised as follows. Section A introduces the interoperable file management example that illustrates the need for mediators to facilitate interoperation between independently-developed components. This example is further used throughout to explain the approach. We also present the formalism we use to specify our inputs, which consists of ontologies to model domain knowledge and annotated labelled transition systems to model the behaviour of components. Section A moves into the solution space, defining the formal methodology to establish semantic correspondences between the operations of component interfaces (a.k.a interface mapping) where ontologies serve as a central reference point for

translation and reconciliation of meaning. We also show how these correspondences can be efficiently computed using constraint programming. Section D describes the algorithms used to generate the mediator by exploring the behaviour of the components and composing the previously produced mappings so as to guarantee that these components interact properly. Section D presents the MICS tool used to synthesise mediators and deploy them in the environment. Section A reports the experiments we conducted to validate our approach using the MICS tool. The results show that our solution covers a large set of mismatches, guarantees the correctness of the mediator, and maintains effective execution time. Section D examines related work. Section A contains an overall analysis and reflections over the features of the approach and its potential enhancements. Finally, Section D concludes the paper.

A.2 Interoperability using Mediators

To highlight the problem of interoperability in distributed systems, we examine the case of two heterogeneous file management systems: WebDAV and Google Docs. We present the models we use to support reasoning about interoperability in distributed systems and outline our approach to the automated synthesis of mediators that enable the successful interoperation of these systems despite differences in their interfaces and behaviours.

A.2.1 The Interoperable File Management Example

The migration from desktop applications to Web-based services is scattering user files across a myriad of remote servers (e.g., Apple iCloud[1], Google Drive[2], and Microsoft Skydrive[3]). This dissemination poses new challenges for users, making it more difficult for them to organise, search, and manipulate their files using their preferred applications, and share them with other users. This situation, though cumbersome from a user perspective, unfortunately reflects the way file management application like many other existing applications has evolved. As a result, users are forced to juggle between a plethora of applications to share their files instead of using their favourite application regardless of the service it relies on or the standard on which it is based.

Among the protocols allowing collaborative management of files, WebDAV (Web Distributed Authoring and Versioning) is an IETF specification that extends the Hypertext Transfer Protocol (HTTP) to allow users to create, read and change documents remotely. It further defines a set of properties to query and manage information about these documents, organise them using collections, and defines a locking mechanism in order to assign a unique editor of a document at any time. Another example is represented by the Google Docs application that lets users create, store, and search Google documents and collections. It also allows users to share and collaborate online in the editing of these documents. These functionalities can be accessed using a Web browser or using the Google proprietary API.

Figure A.1: Connecting a WebDAV client to Google Docs server

Although these two systems offer similar functionalities and use HTTP as the underlying transport protocol, they are unable to interoperate. For example, a user cannot access his Google Docs documents using his favourite WebDAV client (e.g., Mac Finder) as depicted in Figure A.1. This is mainly due to the syntactic naming of data and operations used in each application, and the protocol according to which these operations are performed. Our goal is to synthesise a mediator automatically in order to allow these two components to interact properly. To that end, we need to model the components and their domain so as to reason about their interoperation [116].

[1] http://www.apple.com/icloud/
[2] http://drive.google.com/
[3] http://windows.microsoft.com/skydrive/

A.2.2 Modelling Domains using Ontologies

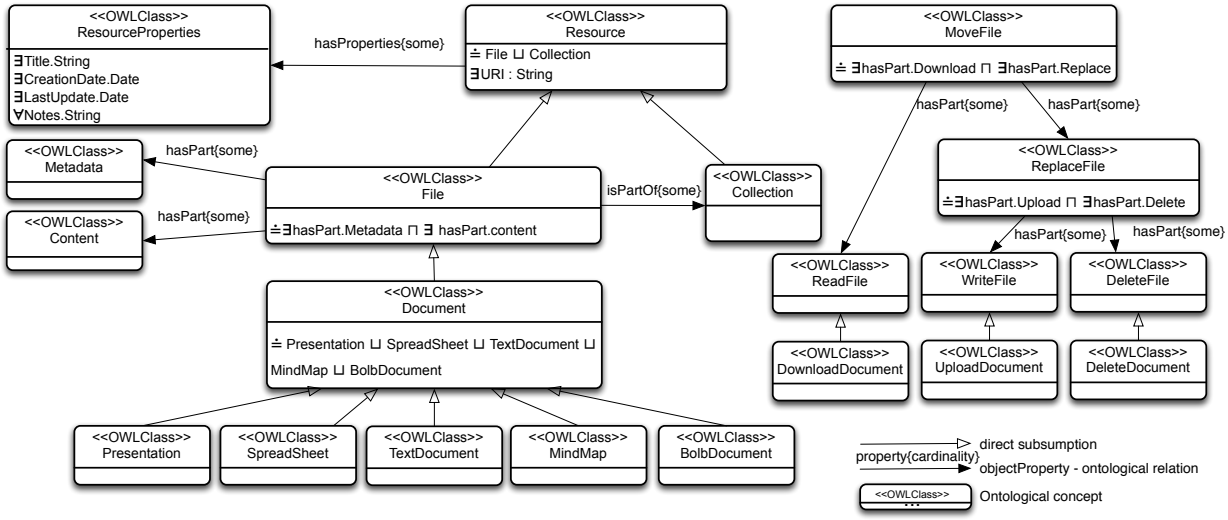

Figure A.2: Extract of the file management ontology

Each application domain has its own vocabulary. This vocabulary has to be modelled explicitly in order to allow computers to conduct automated reasoning about the domain by accessing structured collections of information and sets of inference rules. Ontologies provide experts with a means to formalise the knowledge about the domain as a set of axioms that make explicit the intended meaning of a vocabulary [55]. Hence, besides general purpose ontologies, such as dictionaries (e.g., WordNet[4]) and translators (e.g., BOW[5]), there is an increasing number of ontologies available for various domains such as biology [2], geoscience [99], and social networks [53], which in turn foster the development of a multitude of search engines specially targeted for ontologies on the Web [40].

Ontologies are supported by a logic theory to reason about the properties and relations holding between the various domain entities. In particular, OWL[6] (Web Ontology Language), which is the W3C standard language to model ontologies, is based on description logics. More specifically, we focus on OWL DL, which is based on a specific description logic, $\mathcal{SHOIN}(\mathcal{D})$ [6]. In the rest of the paper, DL refers to this specific description logic. While traditional formal specification techniques (e.g., first-order logic) might be more powerful, DL offers crucial advantages: it has decidable, even efficient reasoning algorithms, yet theory still excels at modelling natural domains.

DL is used to formally specify the vocabulary of a domain in terms of concepts, features of each concept, and relationships between these concepts [43]. DL also allows the definition of complex types out of primitive ones, is able to detect specialisation relations between complex types, and to test the consistency of types. Traditionally, the basic reasoning mechanism in DL is *subsumption*, which can be used to implement other inferences (e.g., satisfiability and equivalence) using pre-defined reductions [6]. In this sense, DL in many ways resembles type systems with some inference mechanisms such as subsumption between concepts and classification of instances within the appropriate concept, corresponding to type subsumption and type inference respectively. Nevertheless, DL is by design and tradition well-suited for application- and domain-specific services [20].

Intuitively, if a concept C is subsumed by a concept D (written $C \sqsubseteq D$), then any instance (also called individual) of C also belongs to D. In addition, all the relationships in which D instances can be involved are applicable to C instances, i.e., all properties of D are also properties of C. Subsumption is a partial order relation, i.e., it is reflexive, antisymmetric, and transitive. The result is that the ontology can be represented as a hierarchy of concepts. These concepts can either be atomic or defined using different operators such as disjunction ($C \sqcup D$), conjunction ($C \sqcap D$), and quantifiers ($\forall R.C$, $\exists R.C$) where C and

[4] http://www.w3.org/TR/wordnet-rdf/
[5] http://BOW.sinica.edu.tw/
[6] http://www.w3.org/TR/owl2-overview/

D are concepts and R is a property. The syntax and semantics of DL operators are summarised in Appendix A, while the interested reader is referred to [6] for further details.

It is important to note that the hierarchy is not created manually. Rather, each concept is given a definition as a set of logical axioms. Then, an ontology reasoner infers a classification, i.e., a hierarchy of concepts. This offers both flexibility and robustness and allows the hierarchy to evolve naturally as new concepts are added. Furthermore, there exist efficient reasoners to automate these inference tasks [41].

Another relation that has been widely investigated is the part-whole relation [4]. We are interested in this relation as it provides a practical means to model aggregations of concepts. Hence, we are not concerned with building a hierarchy of concepts based on the part-whole relation, which is not supported by the wide spread reasoners, but rather by verifying wether a concept subsumes an aggregation of concepts. This can be easily provided by the W3C recommendation for part-whole relation[7] (hasPart) while aggregation can be performed using the conjunction constructor [75]. That is, a concept E is an aggregation of concepts C and D, written $E = C \oplus D$ if and only if both C and D are parts of E. As a result, E can be specified as follows $E = \exists \mathsf{hasPart}.C \sqcap \mathsf{hasPart}.D$.

Example. In the interoperable file management scenario, we build upon the NEPOMUK File Ontology[8] (NFO), which provides vocabulary for describing and relating information elements and operations that are commonly present on desktop applications. Figure A.2 shows an extract of this ontology once the classification has been performed. A Resource has some ResourceProperties and is defined as the union of File and Collection. In DL this would be written: $\mathsf{Resource} \doteq \mathsf{File} \sqcup \mathsf{Collection}$. The *dot* above the "equals" symbol designates a declarative axiom. In addition, a Resource concept has a URI as a data attribute: $\mathsf{Resource} \dot{\sqsubseteq} \exists\ \mathsf{URI}.\mathsf{String}$ and has some properties: $\mathsf{Resource} \dot{\sqsubseteq} \exists \mathsf{hasProperties}.\mathsf{ResourceProperties}$. It can then be inferred that a File is a Resource: $\mathsf{File} \sqsubseteq \mathsf{Resource}$ and that a File also has some properties $\mathsf{File} \sqsubseteq \exists \mathsf{hasProperties}.\mathsf{ResourceProperties}$

In a similar way, when one states that a MoveFile is made up of a ReadFile and a ReplaceFile, which in turn is made up of a WriteFile and a DeleteFile, the reasoner can infer that MoveFile is defined as the aggregation of the three concepts, i.e., $\mathsf{MoveFile} = \mathsf{ReadFile} \oplus \mathsf{WriteFile} \oplus \mathsf{DeleteFile}$ where $=$ is equivalent to a double subsumption. Hence, by giving a formal definition of each concept, the reasoner can infer a hierarchy of concepts.

A.2.3 Modelling Components by Combining Ontologies and Labelled Transition Systems

While ontologies allow domain knowledge to be defined formally, describing knowledge about the software components themselves is equally important. Components usually embody a lot of domain knowledge. Some parts of this knowledge are encoded in the static part of the component in the form of type or operation declarations. Other parts (like for example business rules) are implicitly stored in the dynamic part of the component. To enable automated reasoning about component interaction and perform the appropriate mediation, we must represent explicitly all the domain knowledge implicitly encoded in the application component. Figure A.3 depicts the model of the WebDAV client, which we denote WDAV. The static part is described by the *capability* and the *interface signature* while the dynamic part is described by the *behaviour*.

The capability (Cap) gives a macro-view of the component by specifying the high-level functionality it requires from or provides to its environment [81]. The interface signature (or simply interface) of the component gives a finer-grained description of the operations and data that the component manipulates. We further attach semantic annotations to the interface of the component. Annotations offer a simple technique to include domain knowledge by simply referring to the ontology concepts in the component specification. Augmenting the syntactic description of interfaces with metadata representing logical axioms has been proposed by Borgida and Devanbu [22] and is included in many standards such as OWL-S [81], and SA-WSDL [69].

More precisely, the component interface ($\mathcal{I}$) defines the set of observable *actions* that the component requires/provides from its running environment. It is partitioned into *required* and *provided* actions, with the understanding that required actions are received from and controlled by the environment, whereas

[7] http://www.w3.org/2001/sw/BestPractices/OEP/SimplePartWhole/
[8] http://www.semanticdesktop.org/ontologies/nfo/

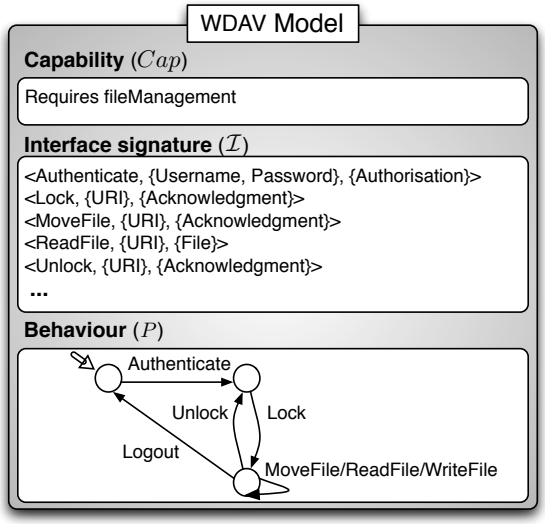

Figure A.3: Model of the WebDAV client

provided actions are controlled and emitted by the component. A required action $\alpha =< op, i, o >$ ($op, i, o \in \mathcal{O}$) calls for an operation op for which it produces some input data i and consumes the output data o. Its dual provided action[9] $\overline{\beta} =< \overline{op}, i, o >$ uses the inputs and produces the corresponding output, as illustrated in Figure A.4. Note that we are not interested in the syntactic description of action elements but rather in the associated concepts in a given ontology $\mathcal{O}$, which provide a precise description of the functional semantics of operations and associated input/output data.

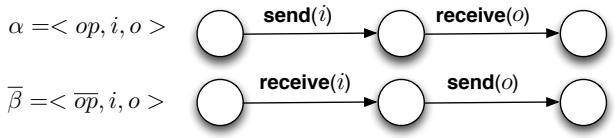

Figure A.4: Required and provided actions

The behaviour of a component specifies its interaction with the environment and models how the actions of its interface are coordinated to achieve the specified capability. We build upon state-of-the-art approaches to formalise component interaction [1, 106] using Finite State Processes (FSP). FSP [78] is a process algebra that has proven to be a convenient formalism for specifying concurrent components, analysing, and reasoning about their behaviours. The actions of the FSP process belong to the component interface. The semantics of FSP is given in terms of Labelled Transition Systems (LTS) [66]. The LTS interpreting a process P is a directed graph whose nodes represent the process states and each edge is labelled with an action belonging to the component interface. $s \xrightarrow{a} s'$ specifies that if the process is in state s and engages in an action $a \in \mathcal{I}$, then it transits to state s'. $s \overset{X}{\Rightarrow} s', X = \langle a_1, ..., a_n \rangle, a_i \in \mathcal{I}$ is a shorthand for $s \xrightarrow{a_1} s_1 ... \xrightarrow{a_n} s'$ denoting that P transits from state s to state s' after it engages in a sequence of actions X. When composed in parallel, processes synchronise on dual actions while actions that are in the alphabet of only one of the two processes can be executed independently. Hence, we assume synchronous semantics. Although the asynchronous semantics can be easily implemented, it is hard to reason about interacting processes under these semantics; in general, properties of the system such as deadlocks are undecidable [25]. The syntax and semantics of FSP are summarised in Appendix A, while the interested reader is referred to [78] for further details.

Finally, we can reasonably assume that the semantic annotations as well as the behavioural information associated within a component are either provided with or derivable from the component interface.

[9] We use the overline as a convenient shorthand to denote provided actions

On the one hand, there are various approaches and standards that emphasise the need and the importance of having such a complete specification [22, 112, 36]. On the other hand, there are more and more advanced learning techniques and tools to support the inference of ontological annotations [10, 94] as well as the extraction of behavioural models [77, 71, 86, 12].

Example. As an illustration, the behavioural specification of the WebDAV client (WDAV) is as follows:

$$
\begin{array}{lcl}
\mathsf{WDAV} & = & (<\mathsf{Authenticate}, \{\mathsf{Username}, \mathsf{Password}\}, \{\mathsf{Authorisation}\} >\!\!\rightarrow \mathsf{P1}), \\
\mathsf{P1} & = & (<\mathsf{Lock}, \{\mathsf{URI}\}, \{\mathsf{Acknowledgment}\} >\!\!\rightarrow \mathsf{P2} \\
 & & \mid <\mathsf{Logout}, \emptyset, \{\mathsf{Acknowledgment}\} >\!\!\rightarrow \mathsf{END}), \\
\mathsf{P2} & = & (<\mathsf{ListFolder}, \{\mathsf{URI}\}, \{\mathsf{FileList}\} >\!\!\rightarrow \mathsf{P2} \\
 & & \mid <\mathsf{ReadFile}, \{\mathsf{URI}\}, \{\mathsf{File}\} >\!\!\rightarrow \mathsf{P2} \\
 & & \mid <\mathsf{WriteFile}, \{\mathsf{File}\}, \{\mathsf{Acknowledgment}\} >\!\!\rightarrow \mathsf{P2} \\
 & & \mid <\mathsf{DeleteFile}, \{\mathsf{File}\}, \{\mathsf{Acknowledgment}\} >\!\!\rightarrow \mathsf{P2} \\
 & & \mid <\mathsf{MoveFile}, \{\mathsf{SourceURI}, \mathsf{DestinationURI}\}, \{\mathsf{Acknowledgment}\} >\!\!\rightarrow \mathsf{P2} \\
 & & \mid <\mathsf{Unlock}, \{\mathsf{URI}\}, \{\mathsf{Acknowledgment}\} >\!\!\rightarrow \mathsf{P1}).
\end{array}
$$

The WebDAV client first authenticates. Before executing any operation, clients have to lock the resource, perform the desired operation and then unlock it again. Finally, they log out to terminate.
The behaviour of the Google Docs service (GDocs) is defined as follows:

$$
\begin{array}{lcl}
\mathsf{GDocs} & = & (<\overline{\mathsf{Authenticate}}, \{\mathsf{Username}, \mathsf{Password}\}, \{\mathsf{Authorisation}\} >\!\!\rightarrow \mathsf{P1}), \\
\mathsf{P1} & = & (<\overline{\mathsf{SetSharingProperties}}, \{\mathsf{URI}, \mathsf{SharingProperties}\}, \\
 & & \{\mathsf{Acknowledgment}\} >\!\!\rightarrow \mathsf{P1} \\
 & & \mid <\overline{\mathsf{ListCollection}}, \{\mathsf{URI}\}, \{\mathsf{DocumentList}\} >\!\!\rightarrow \mathsf{P1} \\
 & & \mid <\overline{\mathsf{DownloadDocument}}, \{\mathsf{URI}\}, \{\mathsf{Document}\} >\!\!\rightarrow \mathsf{P1} \\
 & & \mid <\overline{\mathsf{UploadDocument}}, \{\mathsf{Metadata}, \mathsf{Content}\}, \{\mathsf{Acknowledgment}\} >\!\!\rightarrow \mathsf{P1} \\
 & & \mid <\overline{\mathsf{DeleteDocument}}, \{\mathsf{URI}\}, \{\mathsf{Acknowledgment}\} >\!\!\rightarrow \mathsf{P1} \\
 & & \mid <\overline{\mathsf{Logout}}, \emptyset, \{\mathsf{Acknowledgment}\} >\!\!\rightarrow \mathsf{END}).
\end{array}
$$

It specifies that the Google Docs Service (GDocs) expects a client to authenticate first. Then, the client can list collections of documents or download, upload and delete documents. Finally, it logs out to terminate.

A.2.4 Automated Synthesis of Mediators: Overview

As stated in Section A, although the WebDAV client requires a file management functionality that may be provided by the Google Docs service, their interactions lead to an erroneous execution, namely a *mismatch*. In general, mismatches may occur due to inconsistencies in:

- *The names and types of input/output data and operations.* For example, resource containers are called *folders* in Google Docs and *collections* in WebDAV. Google Docs also distinguishes between different types of documents (e.g., presentation, spreadsheet) whereas WebDAV considers them all as files.
- *The granularity of operations*. The WebDAV protocol provides an operation for moving files from one location to another whereas the Google Docs protocol does not. Still, the move operation can be realised using existing operations to achieve the same task, i.e., it can be carried out by performing the upload, download, and delete operations offered by the GDocs service.
- *The ordering of operations.* The WebDAV protocol requires operations on files to be preceded by a lock operation and followed by an unlock operation. The Google Docs protocol does not have such a requirement. Still, it allows users to restrict or release access to a document by changing its sharing settings. Hence, although the operations of the two systems can be mapped to one another, the sequencing of actions required by the WebDAV protocol is not respected by the Google Docs protocol.

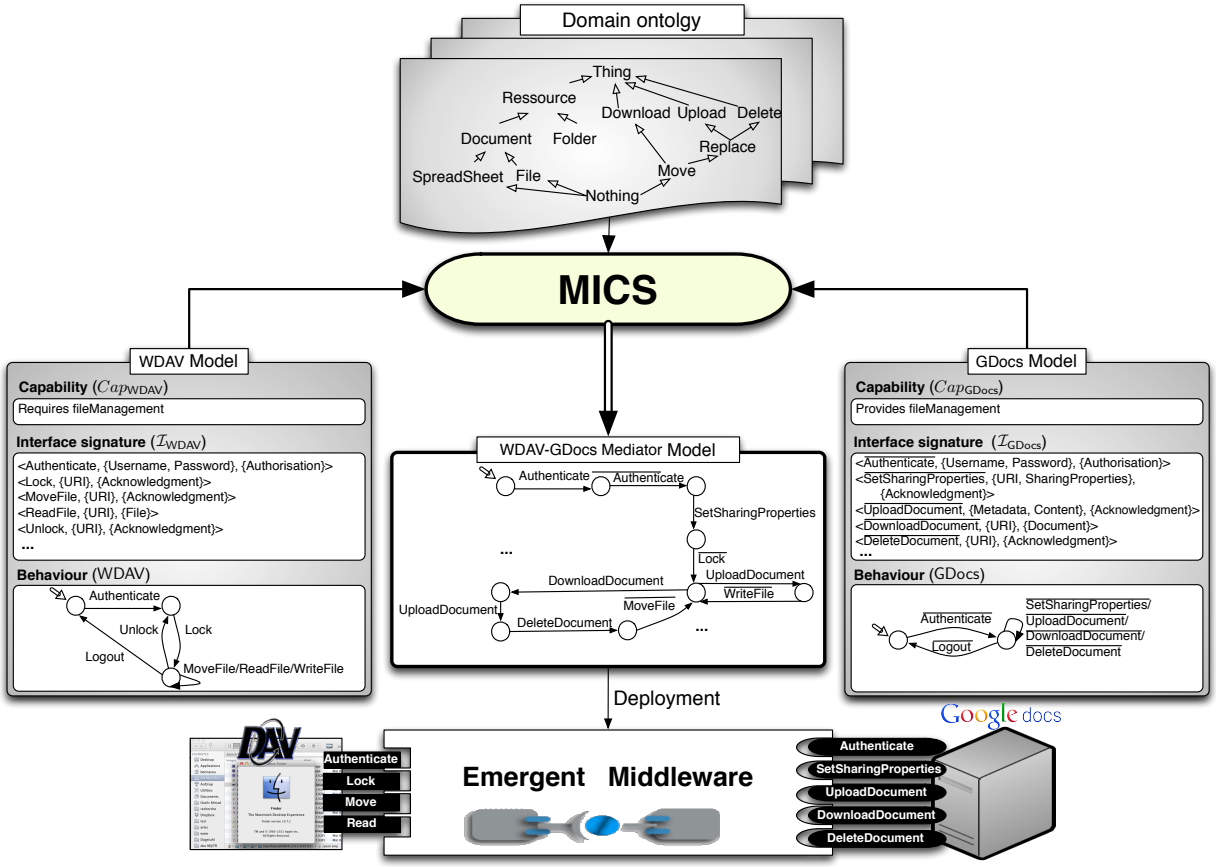

Figure A.5: Overview of our approach to the automated synthesis of mediators

It is the role of the mediator to solve the aforementioned mismatches by compensating for the differences in the data and operations of the components under consideration and coordinating their behaviours to respect the sequence of actions expected by both of them. In order to reason about component interoperation automatically and accordingly generate the appropriate mediator, we rely on the rigorous modelling of both components as defined in Section A. We also use an off-the-self ontology $\mathcal{O}$ to represent domain knowledge and utilise it to reason about the relations holding between the data and operations of the two components. Furthermore, we utilise the ontology $\mathcal{O}$ to verify, prior to any mediation, whether it makes sense for the two components to interact with each other by checking if they are *functionally compatible*, i.e., if the capability required by one component subsumes the capability provided by the other component, in a way similar to capability matching in Semantic Web Services [97].

To enable the interoperation of functionally compatible components, a significant role of the mediator is to convert data available on one side and make it suitable and relevant to the other. This conversion can only be carried out if there exists a semantic equivalence between the actions required by one components and that provided by the other component, that is, interface mapping. The main idea is to use the domain-specific information embodied in the ontology $\mathcal{O}$ in order to select among all the possible combinations of the actions of the components' interfaces only those preserving the semantics and for which automated transformations can be safely realised. In a sense, this is similar to the Liskov Substitution Principle [76]. Nevertheless, by relying on DL, we may define *n*-ary relations between concepts, which allows us not only to substitute an action by another one, but to substitute one sequence of actions by another sequence of actions. Hence, the mapping is based on logical subsumption instead of object types.

One important aspect of interface mappings is that it can be ambiguous, i.e., the same sequence of actions of one component may be mapped to different sequences of actions from the other component. It then becomes crucial to combine the mappings in a way that guarantees that the two components progress and reach their final states without errors that cannot be caught by the type system alone (e.g., deadlock). The gist of the approach is then to generate a mediator, whose LTS is called M, that coordinates these mappings and guarantees that the behaviour of the mediated system, which is represented through the parallel composition the LTSs of both together with that of the mediator is free from deadlocks. Notions such as bisimulation or refinement [58] cannot be applied since they assume the use of the same set of actions (alphabet). Neither is it possible to relabel the processes beforehand since the mappings are not only one-to-one correspondences but rather many-to-many, and also because the same action (sequence of actions) may be translated differently depending on the state in which the system is.

The overall mediation process applied to the interoperable file management example is depicted in Figure A.5. The models of the GDocs and the WDAV components as well as the domain ontology are given as inputs to the MICS tool, which computes the interface mapping and automatically generates the appropriate mediator model. This model is deployed and enacted as an *emergent middleware* [19], which interprets the mediator model and executes the necessary transformations to allow the two components to interact properly.

In the subsequent section, we explain how to compute interface mapping by combining ontology reasoning and constraint programming. Then, in Section D we present how to synthesise the mediator by analysing the behaviours of the components and composing the computed interface mapping accordingly. Finally, in Section D we show how the mediators are deployed as emergent middleware.

A.3 Automated Mapping of Interfaces

The successful interoperation of components is often hampered by *interface mismatches*, which encompass differences in the names and types of the input/output data, and dissimilarities in the granularity of the operations. Establishing the semantic correspondence between the actions of the components' interfaces is a crucial step towards the synthesis of mediators. In this section we first specify the conditions under which such a correspondence, called interface mapping, may be established. It turns out that inferring interface mapping is a combinatorial problem that is efficiently dealt with using constraint programming [100]. We show how existing constraint-programming approaches can be leveraged to compute interface mapping effectively.

A.3.1 Mapping Component Interfaces

Let us consider two components $\mathcal{C}_1$ and $\mathcal{C}_2$ associated with interfaces $\mathcal{I}_1$ and $\mathcal{I}_2$ respectively. An interface mapping $Map(\mathcal{I}_1, \mathcal{I}_2)$ is a relation that states that a required action or sequence of actions X_1 of $\mathcal{I}_1$ can be safely performed by a provided action or a sequence of actions X_2 of $\mathcal{I}_2$. Similarly, $Map(\mathcal{I}_2, \mathcal{I}_1)$ specifies that a required action or sequence of actions X_2 of $\mathcal{I}_2$ can be safely achieved using a provided action or a sequence of actions X_1 of $\mathcal{I}_1$. In other words, mapping $\mathcal{I}_1$ to $\mathcal{I}_2$ consists in finding all pairs (X_1, X_2) where $X_1 = \langle \alpha_i = \langle a_i, I_{a_i}, O_{a_i} \rangle \in \mathcal{I}_1 \rangle_{i=1..m}$ and $X_2 = \langle \overline{\beta_j} = \langle \overline{b_j}, I_{b_j}, O_{b_j} \rangle \in \mathcal{I}_2 \rangle_{i=1..n}$ and such that X_1 *maps to* X_2, denoted $X_1 \mapsto X_2$, if the required actions of X_1 can be safely computed by calling the provided actions of X_2. In addition, this pair is *minimal*, that is, any other pair of actions (X_1', X_2') such that X_1' maps to X_2' would have either X_1 as a subsequence of X_1' or X_2 as a subsequence of X_2'. The interface mapping is then specified as follows:

$$
\begin{array}{l}
Map(\mathcal{I}_1, \mathcal{I}_2) = \\
\{\ (X_1, X_2) | \\
\quad X_1 = \langle \alpha_i = \langle a_i, I_{a_i}, O_{a_i} \rangle \in \mathcal{I}_1 \rangle_{i=1..m} \\
\wedge\, X_2 = \langle \overline{\beta_j} = \langle \overline{b_j}, I_{b_j}, O_{b_j} \rangle \in \mathcal{I}_2 \rangle_{i=1..n} \\
\wedge\, X_1 \mapsto X_2 \\
\wedge\, \nexists (X_1', X_2') \mid\ X_1' = \langle \alpha_i = \langle a_i, I_{a_i}, O_{a_i} \rangle \in \mathcal{I}_1 \rangle_{i=1..m'} \\
\qquad\qquad \wedge\, X_2' = \langle \overline{\beta_j} = \langle \overline{b_j}, I_{b_j}, O_{b_j} \rangle \in \mathcal{I}_2 \rangle_{i=1..n'} \\
\qquad\qquad \wedge\, (X_1' \mapsto X_2') \\
\qquad\qquad \wedge\, (m' < m) \wedge (n' < n) \\
\}
\end{array}
$$

To define the mapping relation ($\mapsto$) precisely, we distinguish between the following cases:

- $m = 1$ *and* $n = 1$. This is a one-to-one mapping, denoted $X_1 \overset{1-1}{\longmapsto} X_2$, and it states that an action required by one component can be safely performed by an action provided by the other component. For example, the $<$ReadFile, {URI}, {File}$>$ action required by WDAV can be performed using the $<\overline{\text{DownloadDocument}}$, {URI}, {Document}$>$ action provided by GDocs.

- $m = 1$ and $n \geq 1$. This is a one-to-many mapping denoted $X_1 \overset{1-n}{\longmapsto} X_2$, and refers to action split/merge, i.e., when an action required by one component is provided by a sequence of actions from the other.For example, the $<$MoveFile, {SourceURI, DestinationURI}, {Acknowledgment}$>$ action required by WDAV can be performed using the $<\overline{\text{DownloadDocument}}$, {URI}, {Document}$>$, $<\overline{\text{UploadDocument}}$, {Metadata, Content}, {Acknowledgment}$>$, and $<\overline{\text{DeleteDocument}}$, {URI}, {Acknowledgment}$>$ actions provided by GDocs.

- $m \geq 1$ and $n \geq 1$. This is the most general case and refers to a many-to-many mapping, denoted $X_1 \overset{m-n}{\longmapsto} X_2$. It is used to specify the case where one sequence of actions corresponds to another sequence of actions.

In the following, we specify the conditions that should be satisfied by the mapping relation ($\mapsto$) in order to guarantee the correct replacement of actions. They express safety properties under which the actions required by one component are consistent with the actions provided by the other component. We first give a formal definition in the one-to-one case, which we extend to the one-to-many and many-to-many case.

A required action $\alpha = \langle a, I_a, O_a \rangle \in \mathcal{I}_1$ maps to a provided action $\overline{\beta} = \langle \overline{b}, I_b, O_b \rangle \in \mathcal{I}_2$ noted $\alpha \overset{1-1}{\longmapsto} \overline{\beta}$, iff:

1. $b \sqsubseteq a$
2. $I_a \sqsubseteq I_b$
3. $I_a \sqcup O_b \sqsubseteq O_a$

The idea behind this mapping is that a required action can be mapped to a provided one if the former supplies all the required input data and the latter provides all the necessary output data, and the required operation is more general than the provided one. This coincides with the Liskov Substitution Principle [76] where ontological subsumption can be used in ways similar to type subsumption. As a result, we can

generate the mapping process M_{1-1} that implements the transformations between actions α and $\overline{\beta}$ as depicted in Figure A.6. M_{1-1} first receives the input of α and translates it into the input required by $\overline{\beta}$. This translation is performed by the f function and can be safely achieved since I_a is subsumed by I_b (Figure A.6-❶). The mapping process M_{1-1} caches the input I_a and uses it together with the received output data O_b to compute the output data expected by α using function g (Figure A.6-❷). This translation is also safe as O_a subsumes the disjunction of I_a and O_a. Put in the FSP format and abstracted from the translation functions, the one-to-one mapping process corresponding the $\alpha \overset{1-1}{\longmapsto} \overline{\beta}$ mapping is as follows: $M_{1-1}(\alpha, \overline{\beta}) = (\beta \rightarrow \overline{\alpha} \rightarrow \text{END})$.

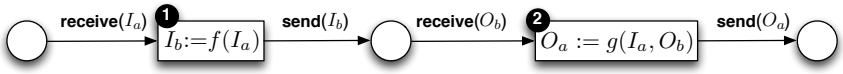

Figure A.6: One-to-one mapping process

Following the same idea, a required action $\alpha = \langle a, I_a, O_a \rangle \in \mathcal{I}_1$ maps to a sequence of provided actions $X_2 = \left\langle \overline{\beta_i} = \left\langle \overline{b_i}, I_{b_i}, O_{b_i} \right\rangle \in \mathcal{I}_2 \right\rangle_{i=1..n}$ noted $\alpha \overset{1-n}{\longmapsto} \left\langle \overline{\beta_1}, ..., \overline{\beta_n} \right\rangle$, iff:

1. $\bigsqcup_{i=1}^{n} b_i \sqsubseteq a$
2. $I_a \sqsubseteq I_{b_1}$
3. $I_a \bigsqcup \left(\bigsqcup_{j=1}^{i-1} O_{b_j} \right) \sqsubseteq I_{b_i}$
4. $I_a \bigsqcup \left(\bigsqcup_{j=1}^{n} O_{b_j} \right) \sqsubseteq O_a$

The first condition states that the required operation a can appropriately be provided using b_i operations, that is the disjunction of all b_i is subsumed by a. The second ensures that the sequence of provided actions can be initiated since the input data of the first action I_{b_1} can be obtained from the input of the required action I_a. The third condition specifies that the input data of each action can be produced out of the data previously received either as input from α or as output from the preceding β_j $(j < i)$. This is necessary since an action can only be executed if its input data is available. The fourth one guarantees that the required output data O_a can be obtained from the set of data accumulated during the execution of all the provided actions.

The associated mapping process M_{1-n} receives the initial input and uses it to initiate the sequence of provided actions (see Figure A.7-❶). Then, M_{1-n} caches the output data produced by performed actions in order to generate the appropriate input of the subsequent ones using translation functions f_i. Once all the operations have been executed, M_{1-n} forwards the output data to the required action in the appropriate format using the translation function g (see Figure A.7-❷). Hence, we represent the mapping process corresponding to $\alpha \overset{1-n}{\longmapsto} \left\langle \overline{\beta_1}, ..., \overline{\beta_n} \right\rangle$ in FSP as follows: $M_{1-n}(\alpha, X_2) = (\beta_1 \rightarrow \beta_2 \rightarrow ... \rightarrow \beta_n \rightarrow \overline{\alpha} \rightarrow \text{END})$.

Following on, a sequence of required actions $X_1 = \langle \alpha_i = \langle a_i, I_{a_i}, O_{a_i} \rangle \in \mathcal{I}_1 \rangle_{i=1..m}$ maps to a sequence of provided actions $X_2 = \left\langle \overline{\beta_j} = \left\langle \overline{b_j}, I_{b_j}, O_{b_j} \right\rangle \in \mathcal{I}_2 \right\rangle_{j=1..n}$ noted $\langle \alpha_1, ..., \alpha_l, ..., \alpha_m \rangle \overset{m-n}{\longmapsto} \left\langle \overline{\beta_1}, ..., \overline{\beta_n} \right\rangle$, iff:

1. $\bigsqcup_{j=1}^{n} b_j \sqsubseteq \bigsqcup_{i=1}^{m} a_i$
2. $\bigsqcup_{i=1}^{l} I_{a_i} \sqsubseteq I_{b_1}$

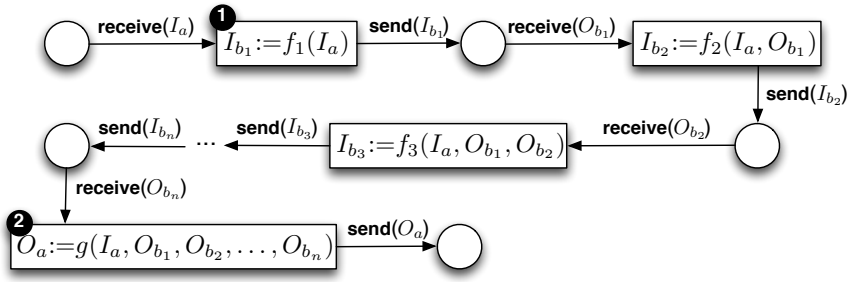

Figure A.7: One-to-many mapping process

3. $\left(\bigsqcup_{j=1}^{l} I_{a_j}\right) \bigsqcup \left(\bigsqcup_{h=1}^{i-1} O_{b_h}\right) \sqsubseteq I_{b_i}$

4. $\forall h \in [1, l[,\ O_{a_h} = \emptyset$

5. $\forall h \in [l, m], \left(\bigsqcup_{i=1}^{h} I_{a_i}\right) \bigsqcup \left(\bigsqcup_{k=1}^{n} O_{b_k}\right) \sqsubseteq O_{a_h}$

The first conditions states that the functionality defined as the disjunction of required by a_i operations can be appropriately provided using b_j operations. The second condition states that the execution of provided actions can be initiated if the necessary input data I_{b_1} can be computed based on the data previously received. The third ensures that the input data of each operation can be acquired by considering the received inputs and the output of the preceding operations. Since we assume synchronous semantics, a required action can only be achieved if its output is available, and analogously a provided action can be executed only its input is available. Still, we can accumulate the data produced by required actions and allow them to progress if they do not require any output. Hence, the fourth condition specifies that the first $l-1$ operations do not require any output and can be executed before the provided actions. Finally, the last condition states that the output of the remaining required actions, i.e., from l to m, can be ascertained by taking into account the previous inputs as well as the outputs generated by the provided actions.

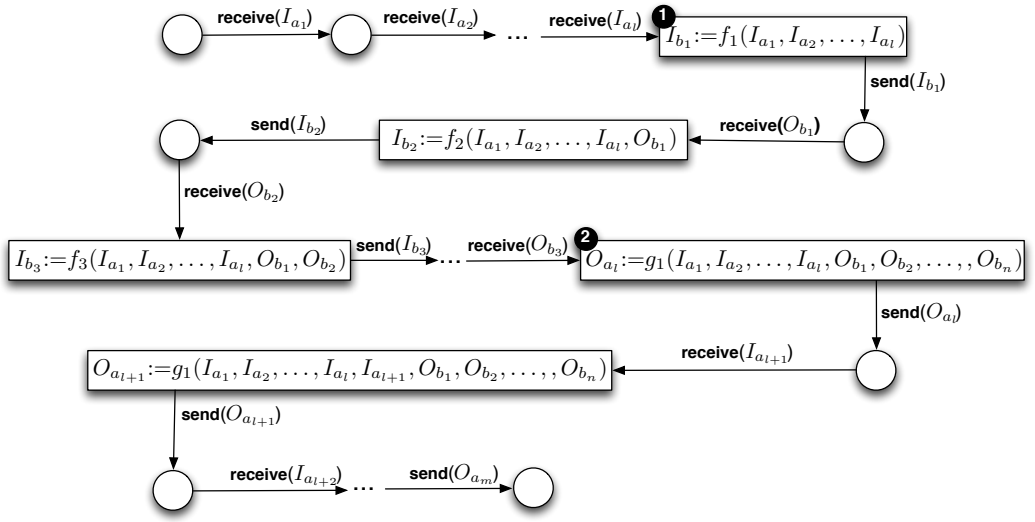

Figure A.8: Many-to-many mapping process

The corresponding mapping process M_{m-n} picks up the data produced by the first l actions and allows the processes to progress, then it uses the data accumulated to produce the input expected by the first

provided action (see Figure B.5-❶). This transformation is safe since I_{b_1} subsumes $\bigsqcup_{i=1}^{l} I_{a_i}$ following the second condition. Then, it chains the provided actions by producing for each operation its expected input in the appropriate format using the translation functions f_i. Once all the provided actions have been performed, it returns the appropriate output to the l^{th} action (see Figure B.5-❷). Then, it continues the execution of the following actions; it receives the input and sends back the suitable output after computing it based on the output data of the b_j operations and the input data of the preceding a_i using the output translation functions g_j. Hence, $M_{m-n}(X_1, X_2) = (\overline{\alpha_1} \rightarrow \overline{\alpha_1} ... \rightarrow \overline{\alpha_{l-1}} \rightarrow \beta_1 \rightarrow \beta_2 \rightarrow ... \rightarrow \beta_n \rightarrow \overline{\alpha_l} \rightarrow \overline{\alpha_{l+1}} ... \rightarrow \overline{\alpha_m} \rightarrow \text{END})$.

To sum up, the conditions states that (i) the functionality offered by the provided actions covers that of the required actions, (ii) each provided actions has its input data available (in the right format) at the time of execution, and (iii) each required actions has its output data available (also in the appropriate format) at the time of execution. Note though that these mappings do not cover all possible mismatches as this would mean that we are able to prove computational equivalence. Still, it covers a large enough set of mismatches with respect to practical and real case studies and other automated approaches for infering interface mapping.

A.3.2 Interface Mapping using Constraint Programming

Mapping interface $\mathcal{I}_1$ to interface $\mathcal{I}_2$ consists in searching, among all the possible pairs of sequences of required actions of $\mathcal{I}_1$ and sequences of provided actions of $\mathcal{I}_2$, those which verify the conditions of the mapping relation specified in the previous section. Furthermore, each pair of sequences of actions is minimal, that is, any other pair verifying the mapping relation would be made up of a sub-sequence of the required or provided actions. As interface mapping is an NP-complete problem (see Appendix A for the proof), we use Constraint Programming (CP) to deal with it effectively. Indeed, CP has proved very efficient when dealing with combinatorial problems [100].

Many arithmetical and logical operators are managed by existing CP solvers. However, although there are some attempts to integrate ontologies with CP [64, 72], none supports ontology-related operators such as subsumption or disjunction of concepts. In order to use CP to compute interface mapping, we need to enable ontology reasoning within CP solvers. Therefore, we propose to represent the ontological relations we are interested in using arithmetic operators supported by existing solvers. In particular, we devise an approach to associate a unique code to each ontological concept and where disjunction and subsumption relations amount to boolean operations.

In the following, we introduce the principles of CP. Then, we formulate the interface mapping as a constrained optimisation problem and show how CP can be used to solve it efficiently.

Constraint Programming in a Nutshell

Constraint programming is the study of combinatorial problems by stating constraints (conditions, properties) which must be satisfied by the solution(s) [100]. These problems are defined as a Constraint Satisfaction Problem and modelled as a triple (X, D, C):

- *Variables*: $X = \{x_1, x_2, ..., x_n\}$ is the set of variables of the problem.
- *Domains*: D is a function which associates to each variable x_i its domain $D(x_i)$, i.e., the set of possible values that can be assigned to x_i.
- *Constraints*: $C = \{C_1, C_2, ..., C_m\}$ is the set of constraints. A constraint C_j is a mathematical relation defined over a subset $x^j = \{x_1^j, x_2^j, ..., x_{n^j}^j\} \subseteq X$ of variables which restricts their possible values. Constraints are used actively to deduce unfeasible values and delete them from the domains of variables. This mechanism is called *constraint propagation*. Efficient algorithms specific to each constraint are used in this propagation.

Solving a constraint satisfaction problem consists in finding the tuple (or tuples) $v = (v_1, ..., v_n)$ where $v_i \in D(X_i)$ and such that all the constraints are satisfied. Thus, CP uses constraints to state the problem declaratively without specifying a computational procedure to enforce them. The latter task is carried

out by a solver. The constraint solver implements intelligent search algorithms such as backtracking and branch and bound which are exponential in time in the worst case but may be very efficient in practice. They also exploit the arithmetic properties of the operators used to express the constraint to quickly check and discredit partial solutions to prune the search space substantially.

For optimisation problems, one needs to define an *objective function* $f : D(x_1) \times ... \times D(x_n) \rightarrow \mathbb{R}$. An optimal solution is then a solution tuple of the constraint satisfaction problem that minimises (or maximises) function f.

We represent interface mapping $Map\,(\mathcal{I}_1, \mathcal{I}_2)$ as a constrained optimisation problem.

- *Variables*: $X = \{x_1, x_2\}$ where x_1 represents a sequence of required actions of $\mathcal{I}_1$ and x_2 represents a sequence of provided actions of $\mathcal{I}_2$.
- *Domains*: $D(x_1)$= $\bigcup_{k=1}^{|\mathcal{I}_1|} P_k(\mathcal{I}_1)$ and $D(x_2)$= $\bigcup_{k=1}^{|\mathcal{I}_2|} P_k(\mathcal{I}_2)$ where $P_k(S)$ denotes the set of k-permutations of the elements of the set S. Indeed, x_1 is a sequence of actions (hence the permutations) of $\mathcal{I}_1$ of length k varying between 1 and the cardinality of $\mathcal{I}_1$, i.e, $1 < k < |\mathcal{I}_1|$. Similarly for x_2.
- *Constraints*: the constraints are defined by the conditions of the mapping relation ($\mapsto$) specified in Section A.
- *Objective function*: the objective function f is based on the length of the sequences of actions, that is $f(x_1, x_2) = |x_1| + |x_2|$

The solutions to the constrained optimisation problem are pairs of action sequences $(\alpha, \overline{\beta}) \in D(x_1) \times D(x_2)$ such that the required actions of α can safely be achieved using the provided actions of $\overline{\beta}$, i.e., $\alpha \mapsto \overline{\beta}$ and $f(\alpha, \overline{\beta})$ is minimal.

Representing Ontological Relations

Our goal is to leverage CP solvers to perform interface mapping as no existing CP solver deals with ontology-based operators. To this end, we define a bit vector encoding of the ontology which is correct and complete regarding the subsumption and disjunction axioms. Correctness means that if the encoding asserts that a concept subsumes another concept or that a concept is a disjunction of other concepts then these relations can be verified in the ontology. Completeness signifies that the subsumption and the disjunction relations specified in the ontology can be verified by the encoding. Specifically, we define the relations $\mathcal{R}_{\sqsubseteq}$ and $\mathcal{R}_{\sqcup}$ such that:

$C \sqsubseteq D \Longleftrightarrow \mathcal{R}_{\sqsubseteq}(C, D)$

$E \doteq C \sqcup D \Longleftrightarrow \mathcal{R}_{\sqcup}(E, C, D)$

We do not distinguish between the aggregation ($\oplus$) and disjunction ($\sqcup$) constructors when computing the interface mapping as they both represent compositions of concepts. The distinction is however maintained at the ontological level, and also within the translation functions of the mapping processes. In the case of disjunction $E \doteq C \sqcup D$, the translation consists in producing an instance of E by assigning to it either an instance of C or an instance of D. While in the case of aggregation $E \doteq C \oplus D$, an instance of E is produced by combining its parts, i.e., both C and D instances, in the appropriate way. Hence, the aggregation relation is also encoded using the $\mathcal{R}_{\sqcup}$ relation as follows:

$E \doteq C \oplus D \Longleftrightarrow \mathcal{R}_{\sqcup}(E, C, D)$

The algorithm for encoding an ontology (Algorithm 1) takes as its input the classified ontology, i.e., an ontology that also includes inferred axioms, and returns a map that associates each concept with a bit vector. More specifically, we first use sets to encode the ontology concepts such that subsumption coincides with subset inclusion and disjunction with set union. Then, we represent the sets using bit vectors of which the size is the number of elements of all sets (Line 26). Each bit is set to 1 if the corresponding element belongs to the set and to 0 otherwise. The type of elements of the sets does not matter, they are just temporary objects used to perform the encoding.

The first step of the encoding algorithm is to assign a unique element to the set that represent each concept (Lines $1-3$). Then, we augment the set of each concept with the elements of the sets associated

Algorithm 1 EncodingOntology

Require: Classified ontology $\mathcal{O}$
Ensure: $Code[]$: maps each concept $C \in \mathcal{O}$ to a bit vector

for all $C \in Concepts(\mathcal{O})$ **do**
  $Set[C] \leftarrow \{\text{NewElement}()\}$
end for
for all $C \in Concepts(\mathcal{O})$ **do**
  for all $Des \in \text{Descendants}(C)$ **do**
    $Set[C] \leftarrow Set[C] \cup Set[Des]$
  end for
end for
$disjunctionAxiomList = \text{Sort}(DisjunctionAxioms(\mathcal{O}))$
for all $A \doteq \bigsqcup_{i=1}^{n} A_i \in disjunctionAxiomList$ **do**
  $\mathcal{D} \leftarrow Set[A] \setminus \bigcup_{i=1}^{n} Set[A_i]$
  for all $d \in \mathcal{D}$ **do**
    $Set[A] \leftarrow Set[A] \setminus \{d\}$
    for all A_i **do**
      $d_i \leftarrow \{NewElement()\}$
      $Set[A_i] \leftarrow Set[A_i] \cup \{d_i\}$
      for all $Asc \in Ascendants(A_i)$ **do**
        $Set[Asc] \leftarrow Set[Asc] \cup d_i$
      end for
    end for
    for all $Des \in \text{Descendants}(A) \mid d \in Set[Des]$ **do**
      $Set[Des] \leftarrow (Set[Des] \setminus \{d\}) \bigcup \left(\bigcup_{i=1}^{n} d_i \right)$
    end for
  end for
end for
$Code[] \leftarrow SetsToBitVectors(Set[])$
return $Code[]$

with the concepts it subsumes, i.e., its descendants (Lines 4−8) since subsumption essentially comes down to set inclusion of the instances of concepts.

We then move to disjunction axioms. We sort the axioms so that each element is made up of simple concepts or preceding concepts in the list (Line 9). For each disjunction axiom $A \doteq \bigsqcup_{i=1}^{n} A_i$, we consider the set $\mathcal{D}$ of elements that belong to the set representing A but which are not included in any of the sets of its composing classes A_i (Line 11). These elements are either the distinguishing element of A or put into A's set by one of its sub-concepts during the previous step. The latter represents the case where a concept is subsumed by the disjunction A but not by any of its individual concepts. To preserve the subsumption, each element $d \in \mathcal{D}$ is split up across all the composing classes A_i. Hence, we first remove it from A (Line 13). Then, we create a new element d_i and add it to A_i's set as well as to the sets of its subsuming concepts (Lines 14−20). We also replace the element d in A's descendants by the new elements it was split up into (Lines 21−23). Finally, we encode the sets using bit vectors where each bit indicates whether or an element belongs to the set(Line 26).

As a result, subsumption can be performed using bitwise *and* as follows:
$C \sqsubseteq D \Longleftrightarrow Code[C] \wedge Code[D] = Code[C]$
and corresponds to the $\mathcal{R}_{\sqsubseteq}$ relation we were looking for. The $\mathcal{R}_{\sqcup}$ relation corresponding to disjunction is represented by bitwise *or*:
$C \doteq \bigsqcup_{i=1}^{n} A_i \Longleftrightarrow Code[A] = \bigvee_{i=1}^{n} Code(A_i),$

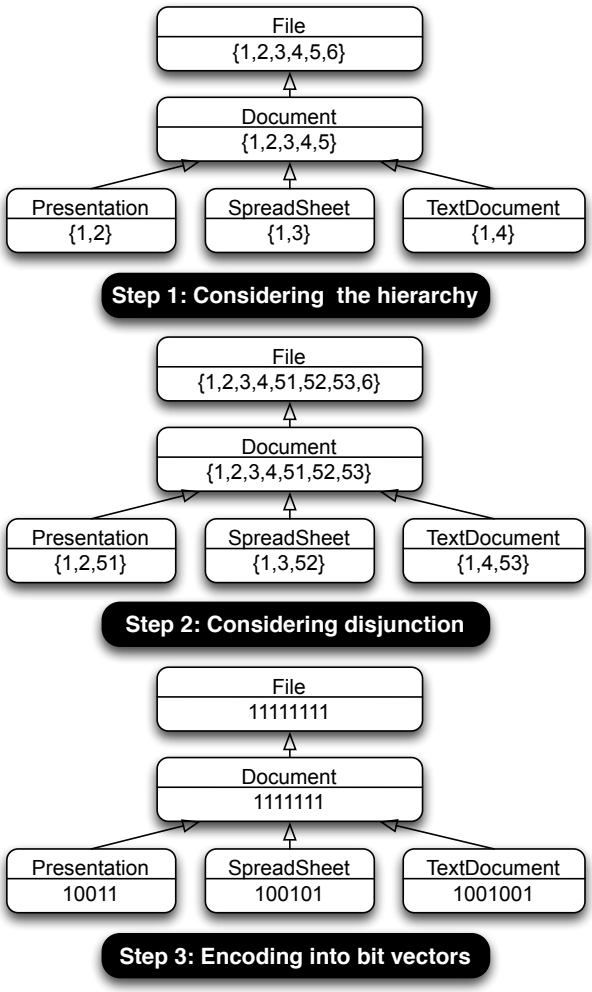

Figure A.9: Ontology encoding example

Example. Let us consider the extract of the file management ontology depicted in Figure A.9. File subsumes Document which is defined as the disjunction of Presentation, SpreadSheet, and TextDocument. During the first step, we associate an element, which we represent as a natural number, to each concept and put it up into its ascendants. The element *'1'* represents the bottom concept $\bot$ subsumed by all concepts. Then, we consider the Document $\doteq$ Presentation $\sqcup$ SpreadSheet $\sqcup$ TextDocument disjunction. The *'5'* element belongs to Document but not to any of its composing concepts, so we split it into three elements (*'51'*, *'52'*, and *'53'*) and assign each of them to the composing elements and to all its ascendants during step 2. Then during step 3, we encode sets as bit vectors. For example, Presentation includes 1 at the position of *'1'*, *'2'*, and *'51'* elements; and 0 at all other positions. The bitwise *and* between the codes of File and Document corresponds to the code of Document ($11111111 \wedge 1111111 = 1111111$), which is equivalent to stating that File subsumes Document. We can then write $\mathcal{R}_{\sqsubseteq}(\text{Document}, \text{File})$. The bitwise *or* between the codes of Presentation, SpreadSheet, and TextDocument is 1111111, which corresponds to the value of Document. We can than write $\mathcal{R}_{\sqcup}(\text{Document}, \text{Presentation}, \text{SpreadSheet}, \text{TextDocument})$.

The encoded ontology is used by the CP solver when computing the interface mapping to check if a constraint is verified. Consider, for example, the action $<$ReadFile, {URI}, {File}$>$ from $\mathcal{I}_{\text{WDAV}}$ and the action $<\overline{\text{DownloadDocument}}, \{\text{URI}\}, \{\text{Document}\}>$ from $\mathcal{I}_{\text{GDocs}}$. When instantiating DownloadDocument $\sqsubseteq$ ReadFile (see Figure A.2) and so in this case the pair $(<$ ReadFile, {URI}, {File}$>$, $<\overline{\text{DownloadDocument}}$, {URI}, {Document}$>)$ belongs to $Map\,(\mathcal{I}_{\text{WDAV}}, \mathcal{I}_{\text{GDocs}})$.

Consider also the $<$MoveFile, {URI}, {Acknowledgment}$>$ action from $\mathcal{I}_{\text{WDAV}}$. and the three actions $<\overline{\text{DownloadDocument}}$, {URI}, {Document}$>$, $<\overline{\text{UploadDocument}}$, {Metadata, Content},{Acknowledgment}$>$, and $<\overline{\text{DeleteDocument}}$, {URI}, {Acknowledgment}$>$ from $\mathcal{I}_{\text{GDocs}}$. First, the condition on the semantics of operations is verified, that is DownloadDocument $\oplus$ UploadDocument $\oplus$ DeleteDocument $\sqsubseteq$ MoveFile. Then, both the download and the delete operations can be performed as they only require URI as input, which is produced by the move operation. The upload operation requires input that can be provided by the download operation since Document $\sqsubseteq$ Metadata $\oplus$ Content (see Figure A.2) and can only be executed after the upload operation. Hence, we can have the following minimal mapping

$<$MoveFile, {URI}, {acknowledgment}$>$
$\mapsto<<\overline{\text{DownloadDocument}}$, {URI}, {document}$>$,
$<\overline{\text{UploadDocument}}$, {metadata, content}, {acknowledgment}$>$,
$<\overline{\text{DeleteDocument}}$, {URI}, {acknowledgment}$>>$

Since there is not any data dependency between the upload and delete operations, there exists another mapping using the same actions in a different order:

$<$MoveFile, {URI}, {acknowledgment}$>$
$\mapsto<<\overline{\text{DownloadDocument}}$, {URI}, {document}$>$,
$<\overline{\text{DeleteDocument}}$, {URI}, {acknowledgment}$>$,
$<\overline{\text{UploadDocument}}$, {metadata, content}, {acknowledgment}$>>$

A.4 Automated Synthesis of Mediators

Given the interface mappings returned by $Map\,(\mathcal{I}_1, \mathcal{I}_2)$ and $Map\,(\mathcal{I}_2, \mathcal{I}_1)$, where all required actions are involved in at least one mapping relation, we need either to generate a mediator M that composes these mappings in order to allow both components, whose behaviours are represented using P_1 and P_2 processes, to coordinate, i.e., the parallel composition $P_1 \| M \| P_2$ successfully terminates by reaching an END state; or we determine that no such mediator exists. If such a mediator exists, then we say that P_1 and P_2 *are behaviourally compatible through a mediator* M, written $P_1 \leftrightarrow_M P_2$.

To generate a mediator, we incrementally build a mediator M by forcing the two processes P_1 and P_2 to progress consistently so that if one requires the sequence of actions X_1, the interacting process is ready to engage in a sequence of provided actions X_2 to which X_1 maps. Given that an interface mapping guarantees the semantic compatibility between the actions of the two components, then the mediator synchronises with both protocols and compensates for the differences between their actions by performing the necessary transformations. This is formally described as follows:

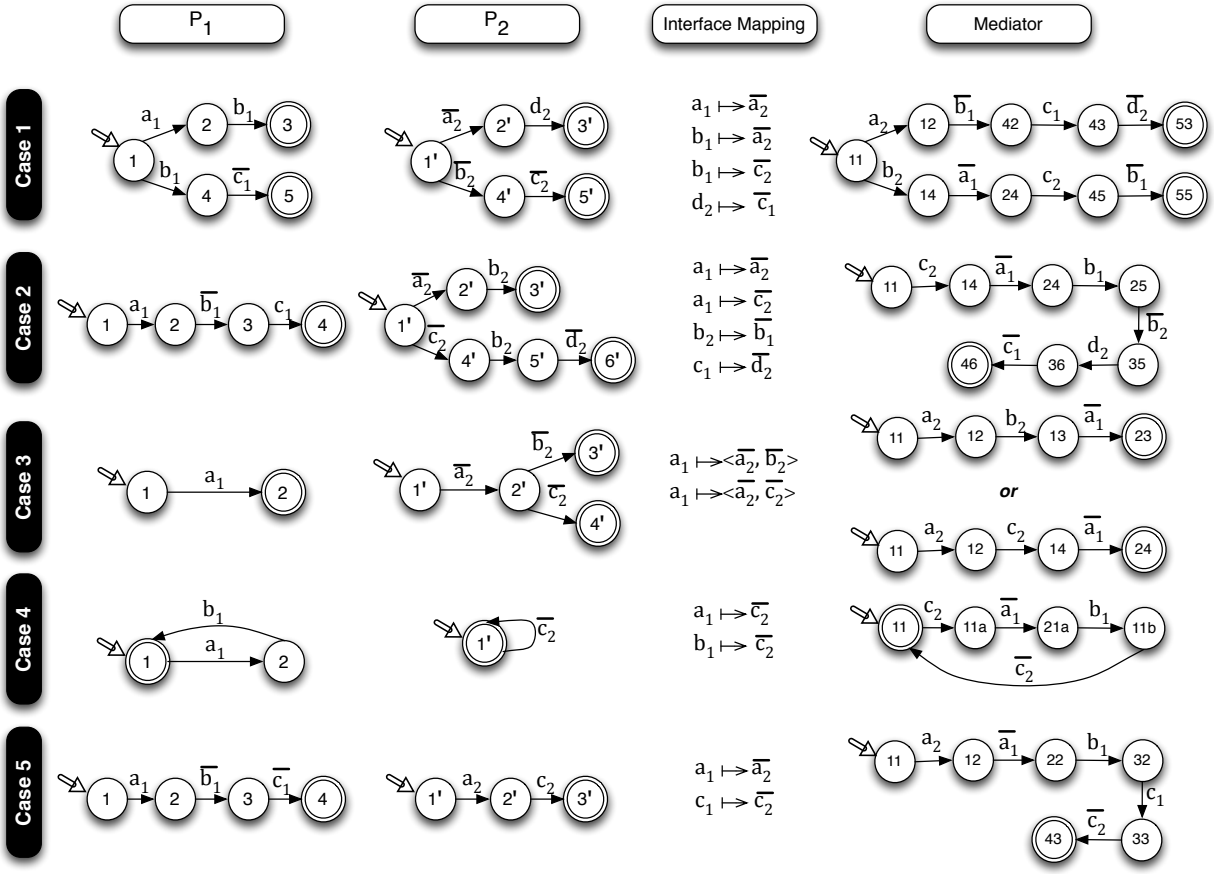

Figure A.10: Mediator examples WITH REAL A/B

if $P_1 \stackrel{X_1}{\Rightarrow} P_1'$ and $\exists\,(X_1, X_2) \in Map\,(\mathcal{I}_1, \mathcal{I}_2)$
such that $P_2 \stackrel{X_2}{\Rightarrow} P_2'$ and $P_1' \leftrightarrow_{M'} P_2'$
then $P_1 \leftrightarrow_M P_2$ where $M = M_{m-n}(X_1, X_2); M'$

Similarly, in the other direction:

if $P_2 \stackrel{X_2}{\Rightarrow} P_2'$ and $\exists(X_2, X_1) \in Map\,(\mathcal{I}_2, \mathcal{I}_1)$
such that $P_1 \stackrel{X_1}{\Rightarrow} P_1'$ and $P_2' \leftrightarrow_{M'} P_1'$
then $P_1 \leftrightarrow_M P_2$ where $M = M_{m-n}(X_2, X_1); M'$

The mediator further consumes the extra provided actions so as to allow protocols to progress; which is specified as follows:

if $P_1 \stackrel{\overline{\beta}}{\rightarrow} P_1'$, and $\exists P_2$ such that $P_1' \leftrightarrow_{M'} P_2$
then $P_1 \leftrightarrow_M P_2$ where $M = (\beta \rightarrow \mathsf{END}); M'$

if $P_2 \stackrel{\overline{\beta}}{\rightarrow} P_2'$, and $\exists P_1$ such that $P_2' \leftrightarrow_{M'} P_1$
then $P_1 \leftrightarrow_M P_2$ where $M = (\beta \rightarrow \mathsf{END}); M'$

Finally, when both processes terminate, i.e., reach an END state, then the mediator also terminates.

$\mathsf{END} \leftrightarrow_{\mathsf{END}} \mathsf{END}$

Note that the interface mapping is not necessarily a function since an action (or a sequence of actions) can be mapped to different actions (or sequences of actions). In the following, we exemplify various cases with which the synthesis algorithm has to deal. These cases, although simple, serve giving an intuitive notion of how to synthesise the mediator.

Case 1: Ambiguous interface mapping but only one mapping is applicable at a given state. In Case 1, b_1 can be mapped to either $\overline{a_2}$ or $\overline{c_2}$. When both processes are at their initial states (1 and $1'$ respectively), the only applicable mapping is $b_1 \mapsto \overline{a_2}$ since P_2 is only able to perform this action. After applying this mapping P_1 moves to state 4, P_2 to state $2'$, and we can create a partial trace of the mediator from 11 to 42. Then, P_2 requires d_2; which maps to the provided action $\overline{c_1}$ and in which P_1 can engage. The result is that P_1 moves to state 5 and P_2 to $3'$, which are both final states. Consequently, we validate the constructed mapping trace and make state 53 final. We then continue exploring the other branch. a_1 can only be mapped to $\overline{a_2}$, which leads P_1 to state 2, P_2 to $4'$ and the partial mediator to 24. Then, b_1 is again required by P_1 but at this point, it can only be mapped to $\overline{c_2}$. Finally, the two processes reach their final states and we can validate the mediator since we successfully explored all outgoing transitions with required actions.

Case 2: Ambiguous interface mapping, multiple mappings applicable at a given state but only one leading to a final state. In Case 1, although an action can be mapped to two actions, only one mapping was applicable at a given state. In Case 2, the required action a_1 can be mapped to either $\overline{a_2}$ or $\overline{c_2}$. Let us assume that the former mapping is selected. P_1 and P_2 move to states 2 and $2'$ respectively, then to 3 and $3'$ after applying the $c_1 \mapsto \overline{a_2}$ mapping. However, P_1 requires an action c_1 whereas P_2 reaches its final state. Consequently, we have to backtrack and select an alternative mapping. At the previous step, only $b_2 \mapsto \overline{b_1}$ was applicable but at the initial state the $a_1 \mapsto \overline{c_2}$ mapping has not been tested. We select this mapping and continue the exploration until we reach states 4 and $6'$, which are the final states of both processes. Hence, it is crucial in each state to keep track of the mappings that have been examined since for each outgoing transition with a required action, we need to select the appropriate mapping that enable both processes to reach their final states.

Case 3: Ambiguous interface mapping, multiple mappings applicable at a given state and all are valid. In Cases 1 and 2, even though a required action can be mapped to different provided actions, only one mapping leads the processes to their final states. In Case 3, two mappings are both valid and allow the processes to reach their final states. We need, however, to select only one of them to generate the mediator since the mediator cannot make a non-deterministic choice on the actions to invoke. The selection of the mapping to use might be motivated by some non-functional property or the length of the mapping but for instance let us assume that we select the first valid mapping. Indeed, we regard the functional concerns as paramount and keep non-functional concerns for future work. The result is that a

correct mediator is not unique. Hence, there are two possible valid mediators: the first translates a_1 to the sequence $\overline{a_2} \rightarrow \overline{b_2}$ while the second translates a_1 to $\overline{a_2} \rightarrow \overline{c_2}$.

Case 4: Multiple required actions mapped to the same provided action. In the previous cases a required action is involved in different mappings. In Case 4, a provided action is involved in different mapping: both a_1 and b_1 maps to $\overline{c_2}$. After performing the a_1 to $\overline{c_2}$ mapping, P_1 moves to a new state that needs to be explored as well while P_2 returns to its initial state. After the second mapping $b_1 \mapsto \overline{c_2}$, P_1 also returns to its initial state but all the outgoing transitions have been treated and it is also a final state, so the mediator is validated.

Case 5: Extra provided action. In the previous cases, the mediator was created so as to map the actions required by one component to the actions provided by the other. The underlying assumption is that the mediator is not able to provide actions by itself, only component do. It may though consume extra provided actions in order to allow the processes to progress as long as the input data of this provided action are available. In Case 5, when P_1 is in state 2 and P_2 in $2'$, c_2 is required but P_1 cannot perform its mapping action $\overline{c_1}$ at this state, so we add the dual action to the mapping trace so as to allow P_1 to progress and get to state 3. In state 3, P_1 can synchronises with P_2 using the mapping $c_1 \mapsto \overline{c_2}$. The idea here is that although the mediator cannot provide an action by itself, it can consume provided action.

These simple cases illustrate the gist of the algorithm we devise to synthesise the mediator (see Algorithm 2). The algorithm starts by checking the basic configuration where both processes reach their final states and where the mediator is the END process (Lines $1-3$). Then it considers the states of both processes and for each enabled required action a, it calculates the list of mappings that can be applied, that is pairs (X_1, X_2) such that X_1 starts with a and P_2 is ready to engage in X_2 (Line 6). It selects one of them and makes a recursive call to test wether it can lead to a valid mediator (Lines $8-9$). If that is the case, it generates the mapping process associated with the selected mapping and puts it in sequence with the returned mediator. Otherwise, it tries another mapping until a valid mapping is found or all the possible mappings have been tested (Lines $7-15$). In the latter case, it checks whether the mediator can bypass a provided action and from there gets a valid mediator, which corresponds to Case 5 in Figure A.10. In this situation, it generates the appropriate process $M_{\overline{\beta}}$ and puts it in sequence with the generated mediator (Lines $16-23$). If the required action cannot be mapped to any action given the states of both processes, the algorithm fails (Lines $24-25$). Otherwise, it adds the new mapping trace to the previously calculated mediator. The algorithm explores all the outgoing transitions labeled with a required action to make sure

Theorem 1 *if $P_1 \leftrightarrow_M P_2$ then the parallel composition $P_1 \| M \| P_2$ is deadlock free. By construction, the mediator is synthesised only if both P_1 and P_2 reach an* END *state. In addition, at any given state s of any of P_1 or P_2, any transition associated with a required action α such that $s \stackrel{\alpha}{\Rightarrow} s'$ is involved in some interface mapping $\alpha \mapsto \overline{\beta}$ and hence have the associated mapping process $Map(\alpha, \overline{\beta})$ ready to engage in with some transition $s_m \stackrel{\overline{\alpha}}{\Rightarrow} s'_m$. Similarly, any transition associated with a provided action $\overline{\beta}$ such that $s \stackrel{\overline{\beta}}{\Rightarrow} s'$ synchronises either with (i) a mapping process $Map(\alpha, \overline{\beta})$ if there exists a mapping $\alpha \mapsto \overline{\beta}$ involving it, or (ii) is an extra provided action, in which case it is associated with $s \stackrel{\beta}{\Rightarrow} s'$ from the $M_{\overline{\beta}}$ process.*

Example. Let us consider the LTSs representing the behaviour of the WebDAV client (WDAV) and that of the GoogleDocs server (GDocs). Figure A.11 depicts an excerpt of each. To simplify the presentation, the actions are defined using the operation concept only. As a result of the interface mapping computation, the Lock and Unlock operations map to SetSharingProperties and the MoveFile operation maps to DownloadDocument, UploadDocument, and DeleteDocument while the last two operations can be executed in any order. When both processes are at their initial states (1 and $1'$ respectively), the only applicable mapping is Lock $\mapsto$ SetSharingProperties since WDAV is only able to perform this action. After applying this mapping WDAV goes to state 2, GDocs remains in state $1'$, and a partial trace of the mediator is created from $11 \rightarrow 11a \rightarrow 21a$. This is similar to Case 4 described in Figure A.10 above.

Then, WDAV can loop on the MoveFile required action, one of the possible mappings is chosen since both are applicable as GDocs loops on the three provided actions, DownloadDocument, UploadDocument, and DeleteDocument. WDAV stays in state 2, GDocs also remains in $1'$ while the mediator is augmented with the trace $21a \rightarrow 21b \rightarrow 21c \rightarrow 21d \rightarrow 21a$. This is similar to Case 3 represented in Figure A.10.

WDAV can also branch on the Unlock operation, which maps to SetSharingProperties and results in the trace $21a \rightarrow 11b \rightarrow 11$ in the mediator. This is again similar to Case 4 described above. Finally, both processes reach their final states and the mediator is successfully created.

Algorithm 2 SynthesiseMediator

Require: P_1, P_2
Ensure: A mediator M
1: **if** $P_1 =$ END and $P_2 =$ END **then**
2: **return** END
3: **end if**
4: $M \leftarrow$ END
5: **for all** $P_i \stackrel{a}{\rightarrow} P'_{i=1,2}$ **do**
6: $mappingList \leftarrow$FindEligibleMappings(a, P_i, P_{3-i})
7: **while** $\neg found$ and $mappingList \neq \emptyset$ **do**
8: $Map(X_1, X_2) \leftarrow selectMapping(mappingList)$ such that $P_i \stackrel{X_1}{\Rightarrow} P''_i$ and $P_{3-i} \stackrel{X_2}{\Rightarrow} P'_{3-i}$
9: $M' \leftarrow$ SynthesiseMediator(P''_i, P'_{3-i})
10: **if** $M' \neq$ **fail then**
11: $found \leftarrow true$
12: $M_{m-n}(X_1, X_2) \leftarrow$GenerateMapProcess$(X_1, X_2)$
13: $M'' \leftarrow M_{m-n}(X_1, X_2); M'$
14: **end if**
15: **end while**
16: **if** $\neg found$ and $\exists \overline{\beta} \mid P_{3-i} \stackrel{\overline{\beta}}{\rightarrow} P'_{3-i}$ **then**
17: $M' \leftarrow$SynthesiseMediator(P_i, P'_{3-i})
18: **if** $M' \neq$ **fail then**
19: $found \leftarrow true$
20: $M_{\overline{\beta}} \leftarrow (\beta \rightarrow$ END$)$.
21: $M'' \leftarrow M_{\overline{\beta}}\,; M'$
22: **end if**
23: **end if**
24: **if** $\neg found$ **then**
25: **return fail**
26: **end if**
27: $M \leftarrow M|M''$
28: **end for**
29: **return** M

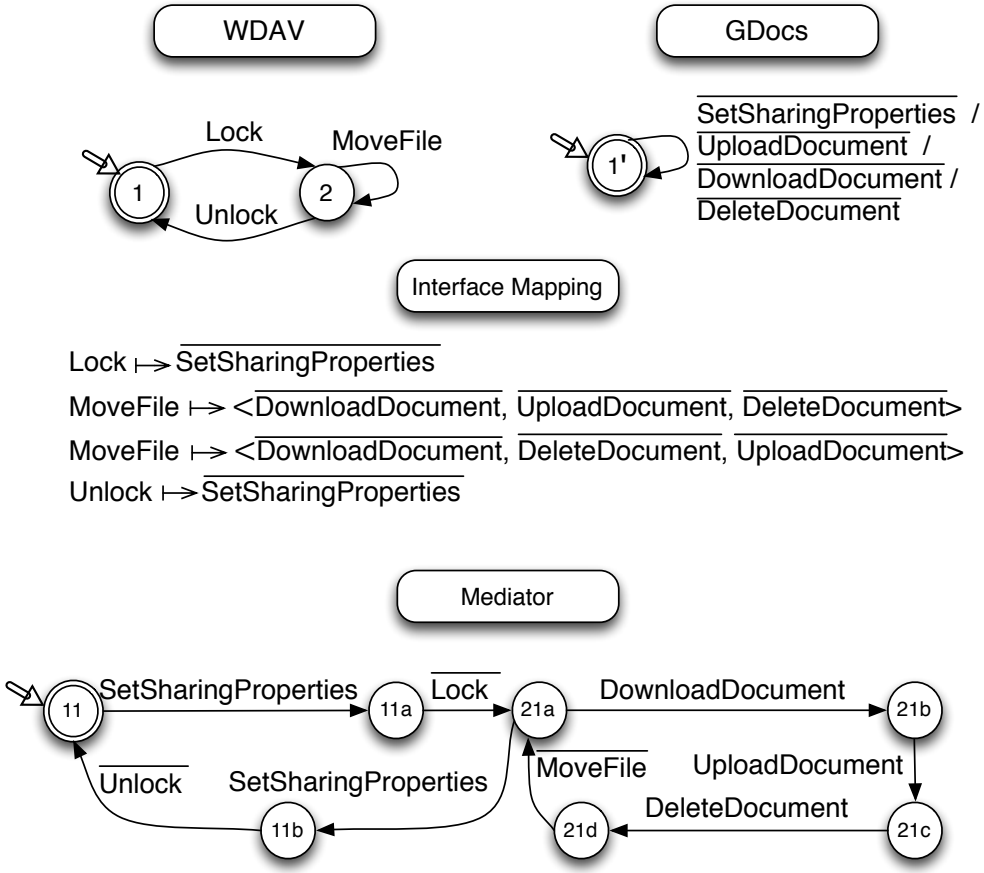

Figure A.11: Synthesis of a WDAV-GDocs (partial) mediator

A.5 Implementation

A.5.1 The MICS tool

In order to validate our approach, we implemented the MICS (Mediator Synthesis to Connecting Components) tool to generate the mediator model automatically. MICS is available at http://www-roc.inria.fr/arles/software/mics/. It is made up of three modules: (i) the ontology encoding module, (ii) the interface mapping module, and (iii) the mediator synthesis module.

The ontology encoding module (see Figure A.12-❶) classifies the ontology using the Pellet reasoner[10]. Pellet is an open-source java library for OWL DL reasoning. Then, it associates bit vectors to the ontology concepts following Algorithm 1.

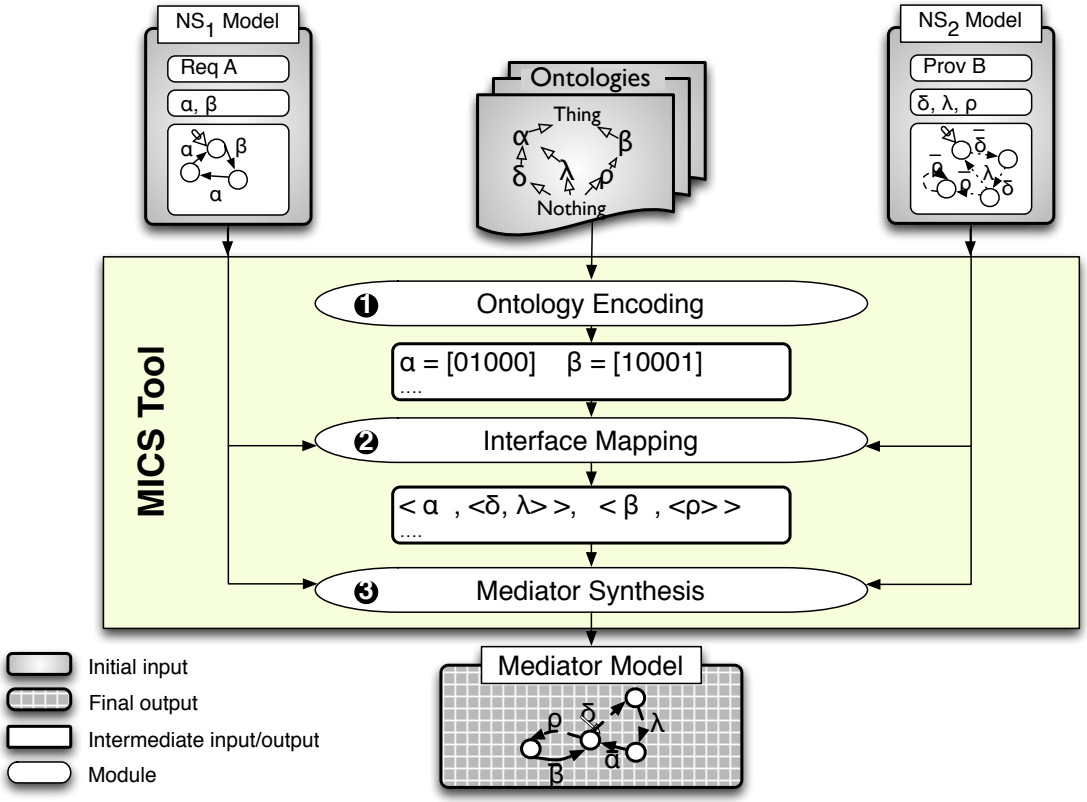

Figure A.12: Overview of the abstract architecture of MICS

The interface mapping module (see Figure A.12-❷) uses Choco[11] to compute the mapping between the interfaces of the components given as an input. Choco is an open-source java library for constraint solving and constraint programming. It is built on an event-based propagation mechanism with backtrackable structures. Choco does not manage ontology relations such as subsumption but, thanks to the bit vector representation of concepts and the associated modelling of constraints, we are able to specify interface mapping as a constrained optimisation problem with operators supported by the Choco library.

The mediator synthesis module (see Figure A.12-❸) relies on the generated mappings to synthesise the mediator according to Algorithm 2.

A.5.2 Enacting Mediators

Once the mediator model has been generated, it needs to be refined and deployed into a concrete artefact so as to realise the specified translations and coordination. This artefact is called an *emergent mid-*

[10] http://clarkparsia.com/pellet/
[11] http://choco.emn.fr/

dleware [19]. The emergent middleware refines the mediator model by incorporating information about underlying middleware and network layers. In particular, the emergent middleware (i) intercepts the input messages, (ii) parses them in order to abstract from the communication details and represent them in terms of actions as expected by the mediator, (iii) performs the necessary data transformations, and (iv) uses the transformed data to construct an output message in the format expected by the interacting component.

Steps i), ii) and iv) are performed using middleware-specific parsers and composers (see Figure A.13). We can either use existing middleware libraries to perform this task or rely on an interpretation framework such as Starlink [27] to generate them at runtime. In the case of the interoperable file management example, we deployed the mediator over an Apache Tomcat[12] container in order to intercept and filter out WebDAV messages from the WebDAV client and used Milton API[13] to parse the messages.

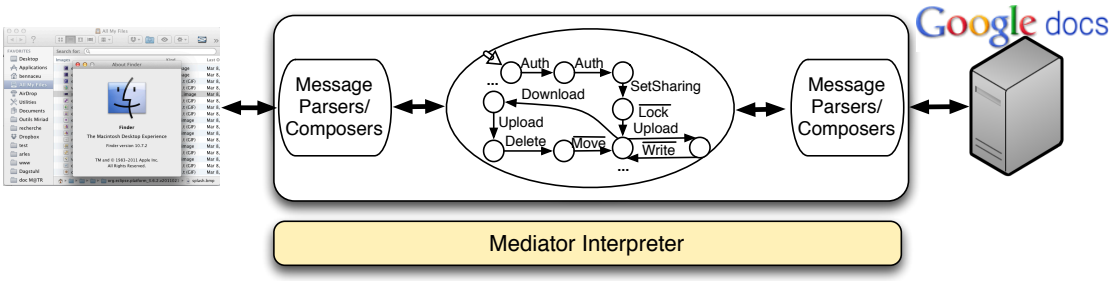

Figure A.13: Emergent middleware

Step iii) needs further computation. Even though, subsumption guarantees the semantic compatibility between concepts, we still need to specify the necessary data transformations in order for the mediator to deal with the syntactic difference between the input/output data. Data mapping is a large and complex problem space [103]. Nevertheless, we should distinguish two cases. In the case of simple types only, the translation is quite straightforward and often consists in simple cast operations. However, in most cases we need to deal with complex data types, e.g., mapping two elaborated XML Schemas. For example, the entry describing a Google Docs pdf file is the following:

```
<entry xmlns:gd="http://schemas.google.com/g/2005"                    1
       gd:etag="'HhJSFgpeRyt7ImBq'">                                  2
  <id>https://docs.google.com/feeds/id/pdf%3AtestPdf                  3
  </id>                                                               4
  <published>2012-04-09T18:23:09.035Z</published>                     5
  <updated>2012-04-09T18:273:09.035Z</updated>                        6
  <app:edited xmlns:app="http://www.w3.org/2007/app">                 7
    2009-06-18T22:16:02.388Z                                          8
  </app:edited>                                                       9
  <title>PDF's Title</title>                                          10
  <content type="application/pdf"                                     11
    src="https://doc-04-20-docs.googleusercontent.com/                12
      docs/secure/m71240...U1?h=1630126&amp;                          13
      e=download&amp;gd=true"/>                                       14
  <link rel="alternate" type="text/html"                              15
    href="https://docs.google.com/ fileview?                          16
    id=testPdf&amp;hl=en"/>                                           17
  <author>                                                            18
    <name>benamel</name>                                              19
    <email>benamel@gmail.com</email>                                  20
  </author>                                                           21
  <gd:resourceId>pdf:testPdf</gd:resourceId>                          22
                                                                      23
```

[12] http://tomcat.apache.org/
[13] milton.ettrema.com/

```
    <gd:lastViewed>2012-06-18T22:16:02.384Z                    24
    </gd:lastViewed>                                           25
    <gd:quotaBytesUsed>108538</gd:quotaBytesUsed>              26
    <docs:writersCanInvite value="false"/>                     27
    <docs:md5Checksum>2b01142f7481c7b056c4b410d28f33cf         28
    </docs:md5Checksum>                                        29
  </entry>                                                     30
```

while the associated WebDAV representation is the following:

```
<a:response>                                                   1
      <a:href>https://docs.google.com/feeds/                   2
        id/pdf%3AtestPdf                                       3
      </a:href>                                                4
      <a:propstat>                                             5
        <a:status>HTTP/1.1 200 OK</a:status>                   6
        <a:prop>                                               7
          <a:displayname>testPdf.pdf</a:displayname>           8
          <d:Author>benamel</d:Author>                         9
        </a:prop>                                              10
      </a:propstat>                                            11
</a:response>                                                  12
```

The bold tags refer to the properties that need to be mapped while the others are either optional or have a default value. Since the focus of our work is on the dynamic synthesis of mediator rather than a novel approach for data transformations, we rely on existing approaches for data mapping that devise different techniques to infer the transformations needed to translate from an XML Schema to another. We refer the interested reader to the complete survey by Shvaiko and Euzenat [103] for a thorough survey and analysis of the approaches that can be applied with this goal in mind. In particular, we rely on the Harmony library[14] to compute the matching between heterogeneous XML Schema and use Apache Dozer[15] to execute the sequence of functions that translates an instance of the source schema into a valid instance for the target schema.

A.6 Case Studies & Validation

This section reports the results of experiments we conducted using MICS to generate a mediator in different case studies. The first case is the Purchase Order Mediation scenario from Semantic Web Service (SWS) Challenge [98]. This example illustrates various mismatches and allows us to compare our approach with similar ones. However, despite the pervasiveness of interoperability issues that we can encounter everyday, very few automated approaches to mediator synthesis are experimented with real applications. Hence, in a second step, we report on the application of our approach to three real world examples in which interoperability is a concern, namely instant messaging, file management, and conference management systems. Finally, we present the experiments we conducted within the CONNECT project using GMES (Global Monitoring of Environment and Security[16]). The aim of this experiments is to illustrate the role of our approach for automated synthesis of mediators in a large scale problem.

A.6.1 Purchase Order Mediation

The Purchase Order Mediation scenario [98] represents a typical real-world problem that is as close to industrial reality as practical. It is intended as common ground to discuss semantic (and other) Web Service solutions and compare them according to the set of features that a mediation approach should support. This scenario describes two commercial systems, *Blue* and *Moon*, that have been implemented using heterogeneous industrial standards and protocols.

[14] http://openii.sourceforge.net/index.php?act=tools&page=harmony
[15] http://dozer.sourceforge.net/
[16] http://www.gmes.info/

Blue initiates the purchase process by sending the customer information and the list of products he want to purchase. Blue expects an acknowledgement which confirm the item that that can be delivered. Its behavioural description is then as follows.

$$\begin{aligned}\text{Blue} = (&<\text{PurchaseOrder}, \{\text{CustomerName}, \text{ShippingAddress}, \text{BillingAddress},\\ &\text{ItemList}\},\{\text{ItemAcknowledgmentList}\} >\rightarrow \text{END}).\end{aligned}$$

Moon uses two backend systems to manage its order processing, namely a Customer Relationship Management system (CRM) and an Order Management System (OMS). To use Moon's service, the client contacts the Customer Relationship Management (CRM) system to retrieve relevant customer details including his identifier. The OMS system also verifies that the customer is authorised to create a new order. The customer can then add items to the newly-created order. Once all items added, the customer closes the order and asks Moon to confirm the delivery of each item. Its behavioural specification is then the following.

$$\begin{aligned}\text{Moon} = (&<\overline{\text{SearchCustomer}}, \{\text{CustomerName}\}, \{\text{Customer}\} >\\ &\rightarrow <\overline{\text{CreateNewOrder}}, \{\text{CustomerID}, \text{ShippingAddress}, \text{BillingAddress}\},\\ &\{\text{OrderID}\} >\rightarrow <\overline{\text{AddLineItem}}, \{\text{OrderID}, \text{ItemID}, \text{ItemQuantity}\}, \emptyset >\rightarrow \text{P1}),\\ \text{P1} = (&<\overline{\text{AddLineItem}}, \{\text{OrderID}, \text{ItemID}, \text{ItemQuantity}\}, \emptyset >\rightarrow \text{P1}\\ &| < \overline{\text{CloseOrder}}, \{\text{OrderID}\}, \{\text{OrderAcknowledgment}\} >\rightarrow \text{P2}),\\ \text{P2} = (&<\overline{\text{ConfirmLineItem}}, \{\text{OrderID}, \text{ItemID}\},\{\text{ItemAcknowledgment}\}>\rightarrow \text{P2}\\ &| <\overline{\text{ConfirmLineItem}}, \{\text{OrderID}, \text{ItemID}\},\{\text{ItemAcknowledgment}\} >\rightarrow \text{END}).\end{aligned}$$

The SWS Challenge provides relevant information about the systems using WSDL and natural language descriptions. The Moon and Blue are provided by the SWS Challenge organisers and can not be altered, although their description may be semantically enriched. The SWS Challenge participants were asked to extend the syntactic descriptions in a way that their approaches can perform the necessary translation tasks.

We build upon the purchase order ontology, which is publicly available as part of the WSDL-S specification[17], and we extend it in order to include concepts representing the operations of Blue's and Moon's interfaces. We attached semantic annotations to each concept used to define the interfaces of Blue and Moon. We specified the behaviour of each system based on the textual descriptions and sequence diagrams given in the SWS Challenge Web site[18] or the related book [98]. Even though there are techniques to associate semantic annotations to the interface description of the system or to extract the associated behaviour, we did it manually since we focus on mediation rather than on the inference of the semantic annotations or the behaviour describing the system. Figure A.14 depicts the mediator LTS generated by MICS, which corresponds to that expected and described by the SWS challenge organisers.

In the following, we compare our solution for automated synthesis of mediator to those presented at the SWS challenge and which tackle the Purchase Order Scenario. First, the WebML solution aims at facilitating the implementation and maintenance of the mediator by following a software engineering process rather than generating the mediator automatically. It provides a framework to facilitate the description of the interacting systems as well as the refinement of the mediator. Our aim is, however, to generate the mediator automatically while relying on existing tools to annotate the interfaces of the systems and define the associated behaviour, and also to execute the mediator.

Compared to the WSMO solution, we use a common ontology to annotate both systems instead of an ontology per system even though we agree that the interacting systems may be defined using heterogeneous ontologies. We believe that in the presence on heterogeneous ontologies, then the synthesis algorithm should deal with the varying confidence about concept relations and manage the imprecision thereof, which is not the case of the WSMO algorithm. Rather, the WSMO solution assumes the mapping to be manually specified at design time. Unlike WSMO, we use the ontology to reason about the semantics of the actions of components' interfaces and compute the mappings automatically. Furthermore, albeit

[17] http://www.w3.org/Submission/WSDL-S/
[18] http://SWSChallenge.org/wiki/

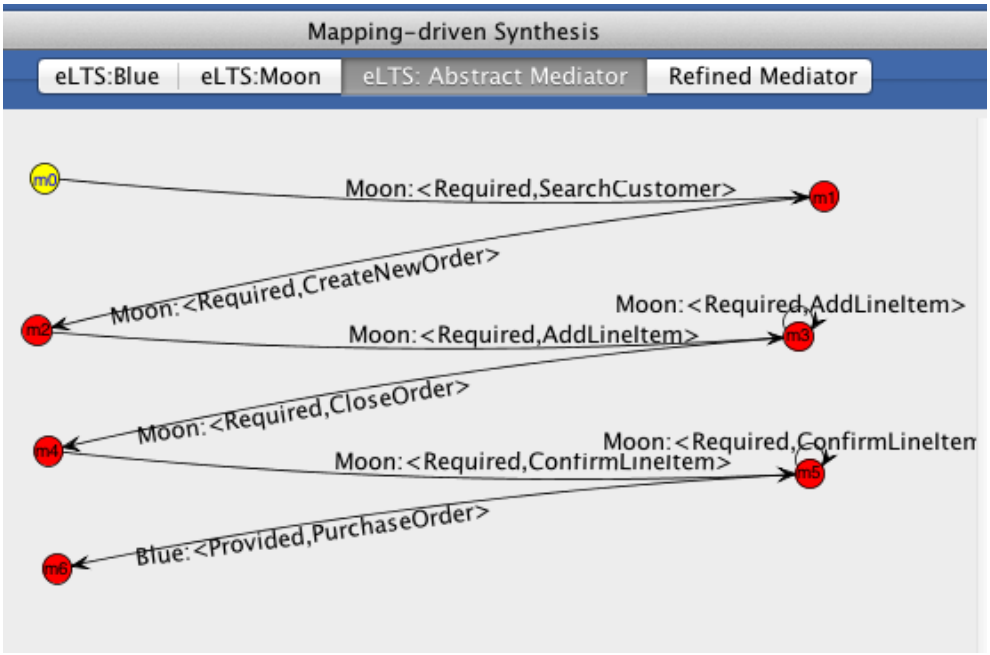

Figure A.14: The purchase order mediator

the schema mapping can boot be proved correct in theory, it turns out that using Harmony to perform schema matching and selecting the data transformations with the higher confidence is sufficient to the XML schema representing the data exchanged between the interacting systems, Blue and Moon. This is probably due to the fact that both systems are defined by the SWS challenge organisers, and hence the semantic difference was minimal. As a consequence, the similarity calculated based on the WordNet, the edit distance, and the structure of XML schema was accurate enough and we did not have to specify the lowering and lifting necessary to perform the required data transformations as is the case with the WSMO solution.

The jABC solution shares with ours the use of process algebra to model the behaviour of Blue and Moon, the use of constraints to ensure safety and consistency of the mediation, and the abstraction/-concretisation to deploy the mediator model using various communication standards. In a first step, the mediator and the constraints were manually defined and jABC is used to verify wether the mediator is correct and consistent with the specified constraints. In a second step, jABC was augmented with the use of an SLTL (Semantic Linear-time Temporal Logic) formula to allow users to describe the goal that need to be fulfilled by the system, based on which they compute the mediator. The use of goal can indeed improve synthesis of mediator but it also raises issue about the management of this goals: does users have to write an SLTL formula to allow use a system? We believe that this might be a quite restrictive requirement. Finally, the declarative approach (devised by the LSDIS lab) is based on adding precondition/effects and using planning techniques to compute the mediator. First the authors specify that it is certainly not enough to consider only precondition/effect (as it is usually done with planning algorithms) but the input/output data too. However, they do not specify how this should be performed. Another drawback of the approach, although not visible from the challenge example, is that it only works with a unique one-to-many mapping and cannot handle interactions where both systems require and provide actions as it is the case in peer-to-peer interactions.

This case study allowed us to compare our approach to similar ones. However, we could not evaluate the mediator we generate since the services are not maintained anymore. Therefore, we used Internet-based application to demonstrate the practical use of our work and evaluate the mediator we synthesise.

A.6.2 Interoperable Instant Messaging

The aim of this case study is twofold. On the one hand, we want to evaluate the MICS tool and the generated mediator with a real application where interoperability is a major concern. In this sense,

The evolution of instant messaging (IM) applications provides an insight into today's communicating applications where different protocols and competing standards co-exist. Popular and widespread IM applications include Windows Live Messenger[19](commonly called MSN Messenger) that is based on MSNP, Yahoo! Messenger[20] that is based on the YMSG protocol, and Google Talk[21] that is based on the Extensible Messaging and Presence Protocol[22] (XMPP) standard protocol. These IM applications offer similar functionalities such as managing a list of contacts or exchanging textual messages. However, a user of MSN Messenger would be unable to exchange instant messages with a user of Google Talk. Indeed, even though XMPP is a W3C standard, many IM (e.g., MSN Messenger and Yahoo! Messenger) continue to use proprietary protocols. Thus, users have to maintain multiple accounts and applications in order to interact with one another. This situation, though cumbersome from a user perspective, unfortunately reflects the way IM – like many other existing applications – has developed. Examples of solutions that address interoperability across IM applications include those providing a uniform user interface (e.g., Adium[23]) and those using a common intermediary protocol (e.g., J-EAI[24]). However, these techniques only provide *ad hoc* solutions to IM interoperability and have difficulty dealing with the changes in the protocols, which are sometimes minimal but require modifying and patching the implemented solution.

To synthesise a mediator that ensure the interoperability of heterogeneous IM protocols, for example MSNP and YMSG, we started by defining the IM ontology (see [11]). We then describe the interfaces of the two IM systems, and annotate them using the IM ontology. Finally, we use the actions of the interface to describe the behaviour of these systems as follows.

$$
\begin{aligned}
\text{MSNClient} &= (<\text{MSN\_Authentication\_Request}, \{\text{UserID}\}, \{\text{Challenge}\} > \\
&\quad \rightarrow <\text{MSN\_Authentication\_Response}, \{\text{Response}\}, \{\text{Authentication\_ok}\} > \\
&\quad \rightarrow \text{ExchangeMsgs}), \\
\text{ExchangeMsgs} &= (<\text{CreateChatRoom}, \{\text{UserID}\}, \{\text{ConversationID}\} > \\
&\quad \rightarrow <\text{JoinChatRoom}, \{\text{UserID}\}, \{\text{Acceptance}\} > \rightarrow \text{P1} \\
&\quad | < \overline{\text{JoinChatRoom}}, \{\text{UserID}\}, \{\text{Acceptance}\} > \\
&\quad \rightarrow < \{\text{ChatRoomInfo}, \emptyset, \{\text{ConversationID}\} > \rightarrow \text{P1}), \\
\text{P1} &= (<\text{InstantMessage}, \{\text{UserID, ConversationID, Message}\}, \emptyset > \rightarrow \text{P1} \\
&\quad | <\overline{\text{InstantMessage}}, \{\text{UserID, ConversationID, Message}\}, \emptyset > \rightarrow \text{P1} \\
&\quad | <\text{MSN\_Logout}, \{\text{UserID}\}, \emptyset > \rightarrow \text{END}).
\end{aligned}
$$

$$
\begin{aligned}
\text{XMPPClient} &= (< \text{XMPP\_Authentication\_Request}, \{\text{UserID}\}, \{\text{Challenge }\} > \\
&\quad \rightarrow <\text{XMPP\_Authentication\_Response}, \{\text{Response}\}, \{\text{Authentication\_ok}\} > \\
&\quad \rightarrow \text{ExchangeMsgs}), \\
\text{ExchangeMsgs} &= (<\text{InstantMessage}, \{\text{SenderID, RecepientID, Message}\}, \emptyset > \rightarrow \text{ExchangeMsgs} \\
&\quad | <\overline{\text{InstantMessage}}, \{\text{SenderID, RecipientID, Message}\}, \emptyset > \\
&\quad \rightarrow \text{ExchangeMsgs} \\
&\quad | < \text{XMPP\_Logout}, \{\text{UserID}\}, \emptyset > \rightarrow \text{END}).
\end{aligned}
$$

Note that some actions, e.g., MSN_Authentication_Request or XMPP_Authentication_Request, are not subject to mediation as they are exchanged with fixed servers rather than between the two applications. As a result, the mediator simply lets them through while it translates the InstantMessage that are exchanged between the clients as depicted in Figure A.15. To deploy the mediator on top of Starlink, we define the

[19] http://explore.live.com/windows-live-messenger/
[20] http://messenger.yahoo.com/
[21] http://www.google.com/talk/
[22] http://www.xmpp.org/
[23] http://adium.im/
[24] http://www.process-one.net

format of the messages of each system using a domain-specific language supported thereof in order to instantiate the appropriate parsers/composers.

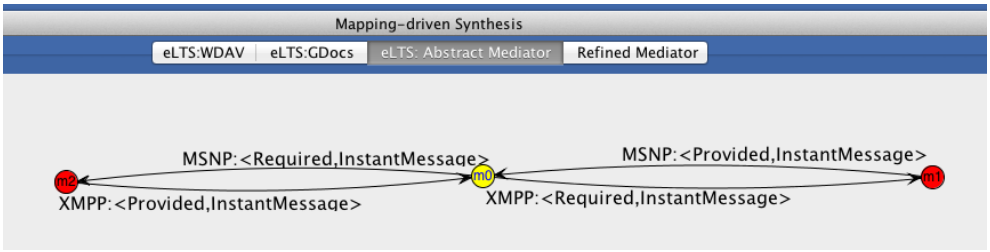

Figure A.15: The instant messaging mediator

To evaluate the performance of the synthesised mediator mediator and measure the overhead the mediator may introduce We deployed the synthesised mediator on the Starlink framework which was running a Mac computer with a 2,7 GHz processor and 8 GB of memory. Figures A.16 provides the average time for message exchange between two IM applications for a message size of 100 characters, considering combinations of MSNP, YMSG, and XMPP. For all combination, we consider both native interactions -if available- and interactions using a synthesised mediator. Besides native interactions between IM services and associated clients, we have native interactions between MSNP and YMSG through the proprietary mediator that is embedded within MSNP and YMSG[25]. The overhead of the mediator in the case of interactions compared to native interactions depends on the types of the instant messaging protocols: while the overhead is negligible for the XML-based XMPP system, it is significant in the case of the binary YMSG protocol. Still, considering the response time experienced by the end-user, the overhead of the mediator execution is acceptable since the largest experiences response time remains close to that of native XMPP communication. It is further worth noticing the performance of MSNP/YMSG interoperability using a mediator that introduces an overhead of 40% compared to the optimised, proprietary interoperability solution.

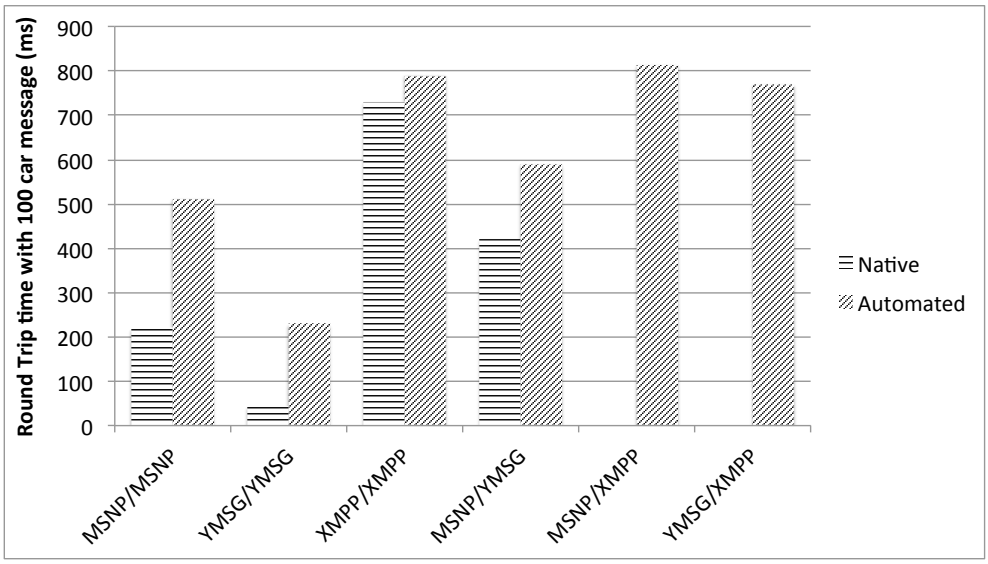

Figure A.16: Time to exchange IM messages in mediated and non-mediated interactions

[25] http://www.microsoft.com/presspass/press/2006/jul06/07-12iminteroppr.mspx

A.6.3 Interoperable File Management

The model of the mediator between WDAV and GDocs synthesised using the MICS tool is illustrated in Figure A.17. We measured the time to perform a simple interaction scenario, which consists in authenticating, moving a file from one folder to another, and listing the content of the two folders. This leads to illustrate the one-to-many mapping as the move operation of WebDAV translates into three Google Docs operations, i.e., download, upload and delete. As for performance measurements, the file is a 4KB text document to lessen the network delay. Figure 4.2 provides the resulting response time, not counting the authentication phase. We can see that the mediator introduces only 17% time overhead compared to WebDAV-only interaction while it keeps performance close to that of GDoc-only interactions.

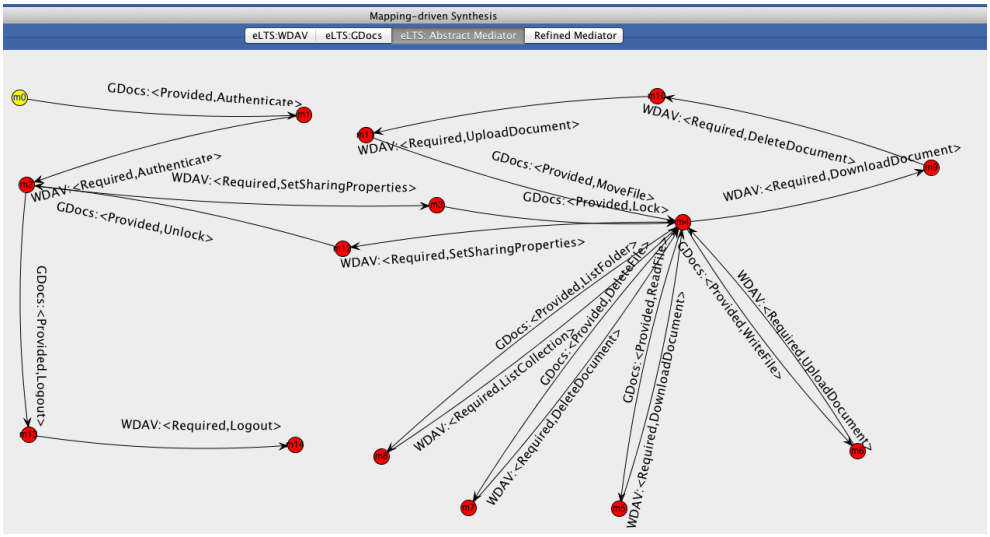

Figure A.17: The file management mediator

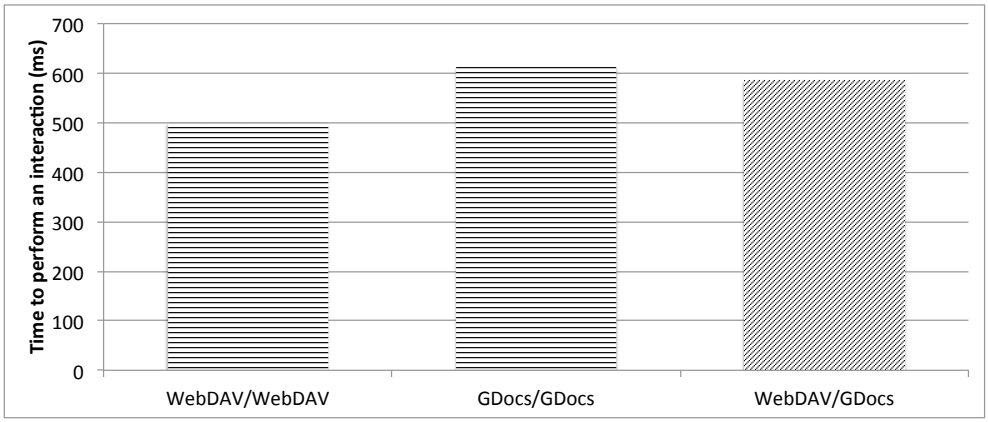

Figure A.18: Time to interact using WebDAV and GDocs in the mediated and non-mediated cases

A.6.4 Global Monitoring for Environment and Security (GMES)

The last case study concerns the area of the Global Monitoring for Environment and Security (GMES[26]). It allows us to evaluate the role of the automated synthesis of mediator in supporting the interoperability lifecycle. In particular, it allows us to integrate with different *enablers*, defined within the CONNECT project,

[26] http://www.gmes.info/

which support universal discovery of networked systems, learning to complete their models, the dynamic synthesis and deployment of a mediator to allow them to interoperate.

GMES is the European Programme for the establishment of a European capacity for Earth Observation. In particular, the emergency management thematic highlights the need to support emergency situations involving different European organisations. In emergency situations, the context is also necessarily highly dynamic and therefore provides a strong example of the need for on-the-fly solutions to interoperability. The target GMES system therefore inevitably involves highly heterogeneous networked systems that interact in order to perform the different tasks necessary for decision making. The tasks include, among others, collecting weather information, capturing video, and getting positioning information of the different devices.

In particular, we concentrate on two examples. The first is concerned with the mediation between a Command and Control centre (C2) that has been designed to interact with some weather station but that have to be mediated in order to use a weather service. The second tackles mediation between the C2 designed to manipulate robots with sensing capabilities (UGV: Unmanned Ground Vehicle) and that need to be mediated in order to manage flying drones (UAV: Unmanned Aerial Vehicle).

We were able to generate mediator so as to enable the command and control centre to get weather information from an external weather service instead of using its own weather stations. In particular while the weather service provide individual information such as temperature or humidity, the C2 uses a unique operation get weather to get . In this case the mediator aggregate the information in a unique message Hence, it is a one to many mapping. We were also able to generate a mediator to . While UGV require the client to authenticate then it can move in the Four cardinal directions, the UAV is required to takeoff prior to any operation. The drone is also required to land before landing. These two operations require only the token to be given. This is the case of extra provided actions. On the other side, the video transmitted by the drone was not in the format expected by the C2, these goes beyond the type question and require manipulation of values and lower level details. An alternative was to use a design-time adaptor, AmbiStream [], to make it possible to convert the video streams.

A.7 Related Work

The problem of mediating interaction models in software has been studied in different contexts, and more notably in the area of software components integration and also recently in Web services. In [61], we survey and analyse the different approaches to mediation and give initial thoughts about an ontology-based approach to the dynamic synthesis of mediators. In this section we present the approaches to mediation from different areas and compare them to the one we propose in this paper.

A.7.1 Architecture-level Mediation

Early work on interoperability in the software architecture field was to identify, catalog and give generic rules and guidelines to developers in order to alleviate architectural mismatches and increase reuse [102]. By defining the formal grounding of connection [1], a more systematic approach to reasoning about interoperability becomes possible. Spitznagel and Garlan [106] define a set of basic transformations to *wrap* connectors in order to reason about interaction between components and further ensure the dependability of their communication. There is also an increasing interest in techniques facilitating integration at runtime. Chang *et al.* [34] propose using *healing connectors* to solve integration problems at runtime by applying healing strategies defined by the developers of the COTS components at design time. Nevertheless, the increasing dynamicity of today's components requires advanced methods to automate the generation of mediators and make it possible at runtime.

A.7.2 Middleware-based Approaches to Mediation

Middleware stands as a conceptual paradigm to connect applications effectively despite heterogeneities in the underlying hardware and software. Nevertheless, each middleware defines a specific message format and coordination model making it difficult (or even impossible) for applications implemented using different middleware to interoperate. Therefore, solutions that bridge applications across different middleware

systems have been devised. Enterprise Service Buses [85] promote loose coupling by implementing protocol translation through a common intermediary protocol. Interoperability platforms (e.g., ReMMoC [54]) enable protocol substitution at runtime by exploiting reflection. However, these solutions require bridges to be created beforehand, which necessitate a substantial development effort and considerable knowledge about the application-domain. Therefore, solutions that reduce the effort of developing bridging solutions have emerged [61]. Specifically, z2z [28] proposes a domain-specific language to describe the protocols to be made interoperable as well as the translation logic to compose them and then generates the corresponding bridge. Starlink [27] enhances this solution by providing a runtime support to the deployment of such bridges. However, these two solutions require the developer to specify the translation to be made and hence to know both protocols in advance whereas in our approach, each protocol is independently specified and the translation is produced automatically.

A.7.3 Semi-Automated Synthesis of Mediators

Existing solutions to reason about the existence and synthesise mediators, assume to be given an abstract specification of the mediator, an adaptation contract [24, 83] or an interface mapping [119]. Among these approaches, pioneering work by Lam [73] uses *image protocols* to reason about the existence of a mediator. An image protocol is derived from a given protocol by partitioning its state set. Two protocols are then compatible if they can be projected onto a common image protocol. However, the developer has to specify the image protocol using an intuitive understanding of the protocols. In their seminal paper, Yellin and Strom [119] propose an algorithm for automated synthesis of mediators based on unambiguous predefined interface mappings. Mateescu *et al.* [83] devise an approach based on process algebra and model checking to synthesise a mediator but require an abstract specification of the way mismatches can be solved, which should not to be ambiguous. Gierds *et al.* [52] rely on a non-ambiguous adaptor specification and use controller synthesis algorithms to generate the mediator. The major difference between these approaches and ours is the consideration of the actions semantics, which allows us to reason about the equivalence of messages and to manage a larger range of mismatches such as one-to-many or many-to-many, which cannot be directly handled considering the syntax of messages alone. Cavallaro *et al.* [33] consider the semantics of data and rely on model checking to identify mapping scripts between interaction protocols automatically. However, they propose that the interface mapping should be performed beforehand so as to align the vocabulary of the processes, but many mappings may exist and should be considered during the generation of the mediator. Nezhad *et al.* [91] consider combined data and behavioural heterogeneity. They devise an approach to identity interface mapping by comparing the XML schemas of individual messages while taking into account their types, i.e., whether a message is sent or received, and considering the associated behavioural protocol. Then, they simulate the interaction protocol in order to filter out the plausible mappings and require the user to check the conflicting mappings. However, the approach focuses on syntactic similarity and does not consider many-to-many correspondence between messages.

A.7.4 Semantic-based Mediation

Web services are processes that expose their interface to the Web so that users can invoke them. Semantic Web Services provide a richer and more precise way to describe the services through the use of knowledge representation languages and ontologies. WSMO [36] defines a description language that integrates ontologies with state machines for representing Semantic Web Services. It also proposes a runtime mediation framework, the Web Service Execution Environment (WSMX), which mediates interaction between heterogeneous services by inspecting their individual protocols and perform the necessary translation on the basis of pre-defined mediation patterns. However, the composition of these patterns is not considered, and there is no guarantee that it will not lead to a deadlock. Vaculín *et al.* [111] devise a mediation approach for OWL-S processes. They first generate all requesters paths, then find the appropriate mapping for each path by simulating the provider process. This approach deals only with client/server interactions and is not able to generate a mediator if many mappings exist for the same action. Finally, Wu *et al.* [118] present an automated approach to process mediation taking into account the semantics of services in terms of their input/output data and preconditions and effects. The approach is based upon planning and requires predefined patterns to manage some process constructs such as choice or loops.

Interoperability has deserved a great deal of work, each providing a little piece. Dealing with both functional and behavioural semantics and doing it automatically is essential. Even though a lot of progress has been make both in understanding and achieving interoperability. It remains an open issue as we can unfortunately see it in our everyday life. We believe that the automation has a great potential and show how to this can be achieved in this paper.

A.8 Future Work

Ontology Heterogeneity. It is crucial to think about ontologies as a means to interoperability rather than universality. Hence, it is often the case that many ontologies co-exist and need to be matched with one another. Ontology matching techniques primarily exploit knowledge explicitly encoded in the ontology rather then trying to guess the meaning encoded in the schemas, as is the case with XML schemas for example. More specifically, while XML schema matching techniques rely on the use of statistics measures of syntactic similarity, ontologies deal with axioms and how they can be put together [103]. This can further benefit from the existence of top-level ontologies such as DOLCE [48] and SUMO [93]. Still, these matching techniques may have some inaccuracy that the synthesis algorithm should deal with. In the future, we aim to extend our model so as to consider the heterogeneity of the ontologies themselves and reasoning about interface mapping under imprecise information.

Partial matching. In our approach, we postulate that each action required by one component must have a compatible action or sequence of actions provided by the other component. This requirement allows us to prove that the mediated system is free from deadlocks but may generate false negatives. It then becomes interesting to authorise partial mediation as long as it achieves the user's or the developer's goal. Indeed, the user may be interested in achieving a specific task only and we can permit interaction between components if we can mediate their behaviour to perform this specific task. To paraphrase, instead of generating the mediator process M such that the parallel composition $P_1 \| M \| P_2$ is deadlock free, we generate M such that the composition satisfies a given goal G, i..e, $P_1 \| M \| P_2 \models G$. We need though to define how the goal is discovered, managed and annotated.

Human in the Loop. In this paper, the synthesis algorithm only relies on the automatically generated interface mapping. Another alternative would be to accept additional transformations by the developers. This raises the issue of the safety of such mapping but could enable a more flexible way to perform the synthesis. As a matter of fact, we can augment the mapping computation an interactive and graphical environment, such as that proposed by ITACA [32], to allow developers to specify additional mappings. This option becomes particularly appealing if there is only a partial matching between the components and the transformations are not exact but rather rely on an intuitive understanding of the components at hand. In the case of an expert, the ontology may also be gradually refined so as to reflect the new understanding of the domain and make a greater degree of automation possible in future systems.

Coping with Changes. Another important property of today's software is their continuous change in order to cope with the market pressures and address evolving user requirements. Our ultimate goal is to make the mediator evolve gracefully as additional knowledge becomes available, system components change, or ontology evolves. In addition, we may use learning techniques to infer the model of the system. While learning significantly improves automation by inferring the model of the component from its implementation, it also induces some inaccuracy that may lead the system to reach an erroneous state. Hence, the system needs to be continuously monitored so as to evaluate the correspondence between the actual system and its model. In the case where the model of one of the components changes, then the mediator should be updated accordingly in order to reflect this change. Another imprecision might also be due to ontology alignment or to partial matching. Hence, incremental re-synthesis would be very important to cope with both the dynamic aspect and partial knowledge about the environment.

A.9 Conclusion

Interoperability is a key challenge in software engineering whether expressed in terms of the compatibility of different components and protocols, in terms of compliance to industry standards or increasingly in terms of the ability to share and reuse data gathered from different systems. The possibility of achieving

interoperability between components without actually modifying their interfaces or behaviour is desirable and often necessary in today's open environments [7]. Mediators promote the seamless interconnection of dynamic and heterogeneous components by performing the necessary translations between their messages and coordinating their behaviour. Our core contribution stems from the principled automated synthesis of mediators. In this paper, we presented an approach to infer mappings between component interfaces by reasoning about the semantics of their data and operations. We then use these mappings to automatically synthesise a correct-by-construction mediator. An important aspect of our approach is the use of ontologies to capture the semantic knowledge about the communicating components. This rigorous approach to generating mediators removes the need to develop *ad hoc* bridging solutions and fosters future-proof interoperability. We believe that this work holds great promise for the future.

Appendix

DL Quick Reference

In this appendix we introduce the standard concept and object property (role) definition. Composite concepts can be built from them inductively with concept constructors and role constructors.

DL Syntax	
A	Atomic concept
$\top$	Top built-in concept
$\bot$	Bottom built-in concept
$\neg C$	Complement
$C \sqcap D$	Conjunction
$C \sqcup D$	Disjunction
$\forall R.C$	Universal quantifier
$\exists R.C$	Existential quantifier

An interpretation $\mathcal{I}$ consists of a non-empty set $\Delta^{\mathcal{I}}$ (the domain of the interpretation function, which assigns to every atomic concept A a set $A^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$ and to every atomic role R a binary relation $R^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$. The interpretation is extended to concept description following inductive definitions.

DL Semantics	
A	$A^{\mathcal{I}} \subseteq \Delta$
$\top$	$\top^{\mathcal{I}} = \Delta$
$\bot$	$\bot^{\mathcal{I}} = \emptyset$
$\neg C$	$(\neg C)^{\mathcal{I}} = \Delta \setminus C^{\mathcal{I}}$
$C \sqcap D$	$(C \sqcap D)^{\mathcal{I}} = C^{\mathcal{I}} \cap D^{\mathcal{I}}$
$C \sqcup D$	$(C \sqcup D)^{\mathcal{I}} = C^{\mathcal{I}} \cup D^{\mathcal{I}}$
$\forall R.C$	$(\forall R.C)^{\mathcal{I}} = \{i \in \Delta \mid \forall j.R^{\mathcal{I}}(i,j) \Rightarrow C^{\mathcal{I}}(j)\}$
$\exists R.C$	$(\exists R.C)^{\mathcal{I}} = \{i \in \Delta \mid \exists j.R^{\mathcal{I}}(i,j) \wedge C^{\mathcal{I}}(j)\}$

FSP Quick Reference

FSP Syntax	
αP	P's alphabet
$a \rightarrow P$	Action prefix
$a \rightarrow P \mid b \rightarrow P$	Choice
$P; Q$	Sequential composition
$P \| Q$	Parallel composition
END	Predefined process, denotes successful termination

The semantics of FSP is defined in terms of Labelled Transition Systems (LTS). The LTS associated with an FSP process P is a quadruple $lts(P) = \langle S, A, \Delta, s_0 \rangle$ where:

- S is a finite set of states,

- $A = \alpha P \cup \tau$ represents the alphabet of the LTS. τ is used to denote an internal action that cannot be observed by the environment of an LTS,
- $\Delta \subseteq S \times A \times S$ denotes a transition relation that specifies that if the process is in state $s \in S$ and engages in an action a, then it transits to state s', written $s \xrightarrow{a} s'$. $s \overset{s}{\Rightarrow} s', X = \langle a_1, ..., a_n \rangle$, $a_i \in \mathcal{I}$ is a shorthand for $s \xrightarrow{a_1} s_1 ... \xrightarrow{a_n} s'$, and
- $s_0 \in S$ indicates the initial state.

In addition, an LTS terminates if there is a state $e \in S$ such that $\nexists (e, a, s_0) \in \Delta$.

Let $lts(P) = \langle S_P, A_P, \Delta_P, s_{0P} \rangle$ and $lts(Q) = \langle S_Q, A_Q, \Delta_Q, s_{0Q} \rangle$ be the LTSs associated with P and Q respectively. The semantics of FSP is as follows.

	FSP Semantics
$a \rightarrow P$	$lts(a \rightarrow P) =< S_P \cup \{s_n\}, A_P \cup \{a\}, \Delta_P \cup \{(s_n, a, s_{0P})\}, s_n >$ where $s_n \notin S_P$
$a \rightarrow P \vert b \rightarrow Q$	$lts(a \rightarrow P \vert b \rightarrow Q) =< S_P \cup S_Q \cup \{s_n\}, A_P \cup A_Q \cup \{a, b\}, \Delta_P \cup \Delta_Q \cup \{(s_n, a, s_{0P}), (s_n, b, s_{0Q})\}, s_n >$ where $s_n \notin S_P$ and $s_n \notin S_Q$
$P; Q$	$lts(P; Q) =< S_P \cup S_Q, A_P \cup A_Q, \Delta_P \cup \Delta_Q, s_{0P} >$ where $s_{0Q} = e_p$ and e_p is terminating state of $lts(P)$
$P \Vert Q$	$lts(P \Vert Q) =< S_P \times S_Q, A_P \cup A_Q, \Delta, (s_{0P}, s_{0Q}) >$ where Δ is the smallest relation satisfying the rules $\frac{P \xrightarrow{a} P', \nexists a \in \alpha Q}{P \Vert Q \xrightarrow{a} P' \Vert Q}$ $\frac{Q \xrightarrow{a} Q', \nexists a \in \alpha P}{P \Vert Q \xrightarrow{a} P \Vert Q'}$ $\frac{P \xrightarrow{a} P', Q \xrightarrow{a} Q', a \neq \tau}{P \Vert Q \xrightarrow{a} P' \Vert Q'}$
END	$lts(\text{END}) =< \{e\}, \{\tau\}, \{\}, e >$ where e is a terminating state

Complexity of Interface Mapping

We prove that interface mapping is NP-complete using polynomial-time reductions from the set cover problem [65]. Recall that in the set cover problem, we are given a set of n elements U and a finite family of its subsets $C = \{S_1, ..., S_m\}$ such that $S_i \subseteq U$ and $\bigcup_{i=1}^{m} S_i = U$, we must find a minimum-cost collection of these subsets whose union is U, i.e., a family of subsets $C' = \{T_1, ..., T_k\}$ such that $T_j \subseteq C$ and $\bigcup_{j=1}^{k} T_i = U$.

The first step is to transform an instance of the set cover problem into an instance of interface mapping. We start by building an ontology made up of disjoint concepts, each of which represents an element of U. Then, the first interface includes a unique required action whose operation is the disjunction of all elements of U and input and output data are void. $\mathcal{I}_1 = \{< \bigsqcup_{x \in U} x, \emptyset, \emptyset >\}$. The second interface $\mathcal{I}_2$ is made up of provided actions only. Each action $\overline{\beta_i}$ is associated with a subset $S_i \in C$ and where the operation of $\overline{\beta_i}$ is the disjunction of S_i's elements and its input and output data are void. Hence, $\mathcal{I}_2 = \bigcup_{S_i \in C} \{< \overline{\bigsqcup_{s \in S_i} s}, \emptyset, \emptyset >\}$.

Since the input and output data are void, interface mapping specifies the pairs $(\alpha, \overline{\beta_1} ... \overline{\beta_k}) \in \mathcal{I}_1 \times (\mathcal{I}_2)^k$ verifying $\bigsqcup_{i=1}^{k} \left(\bigsqcup_{s \in S_i} s \right) \sqsubseteq \bigsqcup_{x \in U} x$. To get a solution to the set cover suffices to pick the subsets associated with the shortest sequence, which can be performed in polynomial time.

Acknowledgments

We gratefully acknowledge discussions with Daniel Sykes, Animesh Pathak and Thiago Teixeira whose insights helped shape this work. We also thank Thierry Martinez for his help on debugging the CP models of the MICS tool. Finally, we thank Richard James for proofreading. This work is carried out as part of the European FP7 ICT FET CONNECT project (http://connect-forever.eu/).

B An Automated Mediation Framework for Cross-Layer Protocol Interoperability

**Amel Bennaceur (Inria), Emil-Mircea Andriescu (Ambientic, Inria),
Roberto Speicys Cardoso (Ambientic), Valérie Issarny (Inria)**

Abstract Today's software systems, and systems of systems, are increasingly developed by composing existing and independently-developed components. Such components often interact using different application protocols and are implemented on top of heterogeneous middleware, which hamper their interoperation. While existing approaches to interoperability consider either application or middleware heterogeneity separately, we argue that in real world scenarios this does not suffice: application and middleware boundaries are ill-defined and solutions to interoperability must consider them in conjunction. In this paper, we propose such a solution, which solves cross-layer interoperability by automatically generating parsers and composers that abstract physical message encapsulation layers into logical protocol layers, thus supporting application layer mediation. Further, whenever possible, the framework automatically synthesizes the appropriate protocol mediators between interacting components based on the reasoning about the components' functional and behavioral semantics. To demonstrate the validity of our approach, we show how the framework solves cross-layer interoperability between existing conference management systems.

B.1 Introduction

Enabling the composition of components regardless of the technology they use and the protocols according to which they interact is a fundamental challenge in complex distributed systems [90]. It has been the focus of a large amount of research, from approaches that identify the causes of interoperability issues and give guidelines on how to address them [49], to approaches that try to automate application of such guidelines [61].

This challenge is exacerbated when heterogeneity spans all component layers, namely *application*, *middleware*, and *network*. At the application layer, components may exhibit disparate data types and operations, and may have distinct business logics. At the middleware layer, they may rely on different communication standards (e.g., CORBA or SOAP) which define disparate data representation formats and induce different architectural constraints [90]. Finally, at the network layer, data may be encapsulated differently according to the network technology in place. Heterogeneity at the network layer has largely been solved by relying on a common standard (i.e., IP - Internet Protocol). However, this is far from being the case for the application and middleware layers.

Middleware provides services that facilitate the connection of components despite the heterogeneity of the underlying platforms, operating systems, and programming languages. However, it also defines a specific message format and coordination model, which makes it difficult (or even impossible) for applications using different middleware to interoperate: an application implemented atop of CORBA cannot communicate with an application developed using SOAP. Furthermore, the evolving application requirements lead to a continuous update of existing middleware tools and the emergence of new approaches. For example, SOAP has long been the protocol of choice to interface Web services but RESTful Web services [45] are becoming increasingly common today. As a result, application developers have to juggle with a myriad of technologies and tools and include *ad hoc* glue code whenever it is necessary to integrate applications implemented using different middleware. Middleware interoperability solutions [61] facilitate this task, either by providing an infrastructure to translate messages into a common intermediary protocol such as in the case of Enterprise Service Buses [35], or by proposing a Domain Specific Language (DSL) to describe the translation logic and to generate corresponding bridges [27]. These solutions, however, provide only an execution framework and still require developers to create or specify the translations needed to make applications interoperate.

Application interoperability solutions, on the other hand, target higher automation and loose coupling. In particular, they rely on intermediary entities, *mediators* [115], to enforce interoperability by mapping the interfaces of components and coordinating their behaviors. There are many approaches to generate mediators based on an *interface mapping* [119, 91]. Interface mapping (also called adaptation contract [24, 84]) establishes the semantic correspondence between operations and data of the considered components. To provide full automation, several approaches extract the interface mapping either by measuring the syntactic similarity of messages [91] or by verifying the semantic compatibility between their operations and data using *ontologies* [33]. Ontologies build upon a sound logic theory to enable reasoning about the domain based on an explicit description of domain knowledge as a set of axioms [6]. Furthermore, application interoperability solutions not only consider the data semantics of components but also their behavioral semantics, which makes the reasoning about interoperability more accurate and the generation of mediators more amenable to automation. They all assume, however, the use of the same underlying middleware (e.g., SOAP) and focus on dealing with mediation at the application layer only.

Hence, even though there exists a plethora of interoperability solutions, they are inappropriate for one of the two following reasons: either (i) they deal with application heterogeneity and generate corresponding mediators but fail

to deploy them on top of heterogeneous middleware, or (ii) deal with middleware heterogeneity while assuming the same application atop and rely on developers to provide all the translations that need to be made. We argue that seamless interoperation is still an open issue because it is a cross-cutting concern and interoperability solutions must consider both application and middleware layers. On the one hand, the application layer provides the appropriate level of abstraction to reason about interoperability and automate the generation of mediators. On the other hand, the middleware layer offers the necessary services for realizing the mediation by selecting and instantiating the specific data structures and protocols. In this paper, we propose to reconcile these two visions and devise a unified approach to deal with interoperability at both application and middleware layers.

In [61], we gave preliminary thoughts about the necessity of mediating components from application down to middleware layers. We also emphasized the role of ontologies in supporting semantic interoperability and sustaining automated mediation in complex distributed systems [19]. In this paper, we use ontologies to support the automated synthesis of mediators at the application layer and we introduce *Composite Cross-Layer (CCL) parsers and composers* to handle cross-layer heterogeneity and to provide an abstract representation of the application data exchanged by the interoperating components. More specifically, we associate the data embedded in messages with annotations that refers to concepts in a domain ontology. As a result, we are able to reason about the semantics of messages in terms of the operations and the data they require from or provide to one another and automatically synthesize, whenever possible, the appropriate mediators. These mediators translate the data exchanged between the interacting components and coordinate their behaviors so as to guarantee their interoperation. Our approach assumes that all the underlying heterogeneous middleware use the request/response interaction paradigm. More specifically, our contribution is threefold:

- *CCL parsers and composers.* We devise an approach for the synthesis of parsers and composers able to process messages sent or received by applications implemented using different middleware, and to describe them using an abstract and uniform representation. Since components are built using different protocols and data, we introduce a high-level specification for assembling atomic parsers and composers, which in turn allows us to: (i) reuse implementations of parsers and composers for standard protocols (e.g., HTTP, SOAP, CORBA), (ii) easily integrate with interface-description and serialization languages (e.g. WSDL, XSD, ASN.1), and (iii) integrate with format-specific reverse-engineering tools (e.g., XML learning). The parsers analyze the messages received on the wire in order to extract only meaningful information, i.e., data related to interaction and hence relevant for mediation. The composers receive the mediated messages in the abstract representation, refine them by incorporating the middleware details, and send them to the interacting component in the expected format. Based on a high-level specification of messages, we synthesize a CCL parser and composer by combining legacy atomic parsers and composers and provide an abstract description of the component interface. This abstract description specifies the syntax of data using XML schemas, and the semantics of the operations together with the associated input/output data using references to concepts in a given domain ontology.
- *Abstract mediators.* We reason about the relations between the operations and data of the interacting components using a domain ontology and infer the mapping that guarantees that operations from one component can safely be performed using operations from the other component. Then, we analyze the component's behavioral specifications so as to compute, if possible, the mediator that composes the computed mappings in a way that guarantees that the components interact without errors, e.g., deadlock.
- *Mediation Framework.* We demonstrate the validity of our approach by implementing a prototype tool to synthesize CCL parsers/composers and abstract mediators. Then, we use it to achieve interoperability between distinct existing conference management systems that run on top of heterogeneous middleware.

The rest of the paper is organized as follows. Section B introduces the conference management scenario and the challenges to interoperability that it poses. This scenario is used throughout the paper to motivate and illustrate the approach. Section B describes the generation of cross-layer parsers and composers. Section B presents the technique used to synthesize mediators that transform abstract messages returned by parsers to fit the expectations (in terms of data type and behavior) of the interacting components. Section D gives an assessment of the approach. Section D examines related work. Finally, Section D concludes the paper and discusses future work.

B.2 Motivating Scenario

Conference management systems provide various services such as ticketing, attendee management, and payment to organize events like conferences, seminars and concerts. Event organizers usually rely on existing and specialized conference management systems to prepare their events as they generally offer a better solution compared to building their own system. To interact with conference management systems, organizers have the possibility of creating custom applications that leverage the system's API to automate certain aspects of conference organization. However,

depending on the conference they are in charge of, organizers may have to interact with multiple conference management system providers. Conference management systems often offer different APIs and are implemented using different middleware technologies.

In this paper we consider the case of two existing conference management systems: Amiando[1] and Regonline[2]. The complete description of both systems is beyond the scope of this paper as they define more than 50 operations each. For the sake of clarity, we consider the following simplified example scenario that strictly follows the publicly available APIs of the two services: a client is looking for conferences that include some keywords in their title, and then analyzes conference information (such as dates or registration fees). In Amiando, clients have to send an `EventFind` request containing the keywords to search for. Every Amiando client is given a unique and fixed `ApiKey` that must be included in every interaction with the Amiando service. The `EventFind` response includes a list of conference identifiers. To get the information about a conference, clients issue an `EventRead` request with the event identifier as a parameter. In Regonline, clients must first invoke the `Login` operation in order to get an `ApiToken`, which must be included in the following requests. After that, the client sends a `GetEvents` request which includes a `Filter` argument specifying the keywords to search for. The client gets in return the list of conferences matching the search criteria including their details.

Both Amiando and Regonline are based on the request/response paradigm, i.e., the client issues a request which includes the appropriate parameters and the server returns the corresponding response. However, Amiando is developed according to the REST architectural style, uses HTTP as the underlying communication protocol, and relies on JSON[3] for data formatting. On the other hand, Regonline is implemented using SOAP, which implies using WSDL[4] to describe the application interface, and is further bound to the HTTP protocol.

Seen at some high-level of abstraction the Amiando client should be able to use the Regoline service as the former requires some functionality that can be provided by the latter. Unfortunately, this is not the case in practice and the Amiando client fails to interoperate with the Regonline service. Let us take a closer look to the causes of such a failure.

Application-layer interface and behavioral mismatches To cooperate, components have to agree on the syntax and semantics of the operations they require or provide together with the associated input and output data. However, the same concepts (e.g., conferences, tickets, and attendees) may be expressed using different data types. To enable the components to interoperate, the data need to be converted in order to meet the expectations of each component. For example, to search for a conference with a title containing a given keyword, the Amiando client simply specifies the keyword in the title parameter, which is of type `String`. The Regonline `GetEvents` operation has a `Filter` argument used to specify the keywords to search for and which is also of type `String`. However, the Regonline developer documentation specifies that this string is in fact a C# expression and can contain some .NET framework method calls (such as `Title.contains(''keyword'')`), which is incompatible with the Amiando search string.

The granularity and sequencing of operations is also very important. For example, the `GetEvents` operation of Regonline returns a list of conferences with the corresponding information. To get the same result in Amiando, two operations need to be performed. First, we perform `EventFind` to get a list of conference identifiers. Then, for each element of the list we call the `EventRead` operation with the identifier as a parameter to get information about the conference.

Middleware-layer mismatches Amiando is based on REST while Regonline is based on SOAP. Messages generated by both systems are incompatible and must be translated to allow them to interoperate. Moreover, the mechanisms provided by each middleware to describe the application interface are different: while SOAP-based Web Services rely on a standard interface description language (WSDL) to describe operations, there is no standard description language for RESTful services, although JSON is widely used, and in particular by Amiando.

Cross-layer data mismatches Even though application and middleware layers are conceptually isolated, in real world scenarios the boundaries between both layers tend to disappear, mixing application and middleware data across multiple layers. This confusion is caused by multiple factors such as performance optimizations, simplified development or bad design decisions. For example, the `Login` operation of Regonline returns an `ApiToken`, which is application data. However, instead of including this token in the argument of subsequent operations, it is inserted in the HTTP messages as an optional header field, which is part of the middleware. In their seminal paper [49], Garlan *et al.* insist on the necessity to make explicit any assumption about interaction in order to make the integration

[1] http://developers.amiando.com/
[2] http://developer.regonline.com/
[3] http://www.json.org
[4] http://www.w3.org/TR/wsdl

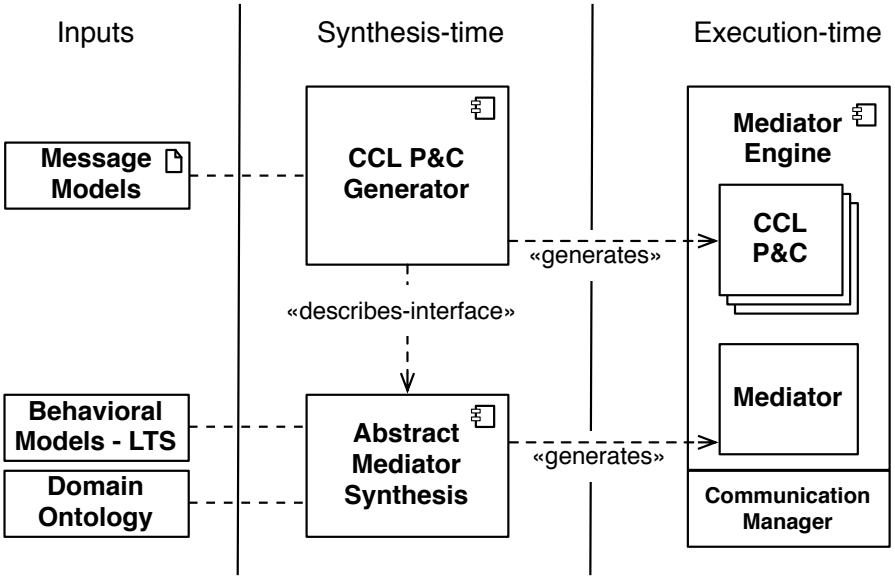

Figure B.1: Mediation framework architecture

feasible and, possibly, automate it. Interoperability solutions should provide a means for developers to describe how the application configures the middleware.

This example, although simple, demonstrates many problems that are faced by developers, and suggests why existing interoperability approaches still fall short in achieving component interoperation automatically. To solve these issues, we propose a framework for cross-layer protocol interoperability that addresses the heterogeneity of the interacting components at both application and middleware layers, and that automates, to a large extent, the generation of parsers, composers and the synthesis of mediators.

Figure B.1 depicts the architecture of the automated mediation framework. The *Composite Cross-Layer Parser & Composer Generator (CCL P&C Generator)* uses the message models to synthesize parsers and composers that consider both application and middleware layers. The *Abstract Mediator Synthesis* component relies on the interface descriptions provided by the CCL P&C Generator to obtain the syntactic description and the ontology-based annotations of the operations required or provided by the interacting components together with the associated input/output data. Using a domain ontology, the Abstract Mediator Synthesis component infers the appropriate mapping between the interfaces of the components. Then, it analyzes their behavioral models in order to generate a mediator that allows them to interact properly. Note that the behavioral specifications of components are either specified by the developer or automatically extracted using existing automata learning techniques, especially those devised within the CONNECT project [12]. Then, at the execution phase, the *Mediator Engine* enables the components to interoperate by executing the generated mediator and relying on the synthesized parsers and composers to deliver the messages in the expected format. Section B details parser and composer generation, while Section B focuses on mediator generation.

B.3 Cross-Layer Parsers and Composers

Whenever components must communicate using heterogeneous message formats, *Parsers* and *Composers* (P&C from now on) are required to interpret messages from one component and to generate messages in a format that the other component understands. Most approaches that enable processing heterogeneous message formats require the specification of P&Cs using a supporting DSL, e.g., ABNF[5][46, 29, 23]. This technique presents three main weaknesses: (i) the workload required for the specification becomes overwhelming when protocols present complex message structures, (ii) all message encapsulation layers are handled at once (thus restraining the reuse of parsers and composers), and (iii) it is limited by the expressiveness and suitability of the DSL with respect to the type of encoding (e.g., text, XML, BER). In this section, we introduce a new approach to generate message P&Cs, which is better suited for the complexity of messages exchanged in real world applications.

[5] Augmented Backus-Naur Form: http://www.ietf.org/rfc/rfc2234.txt

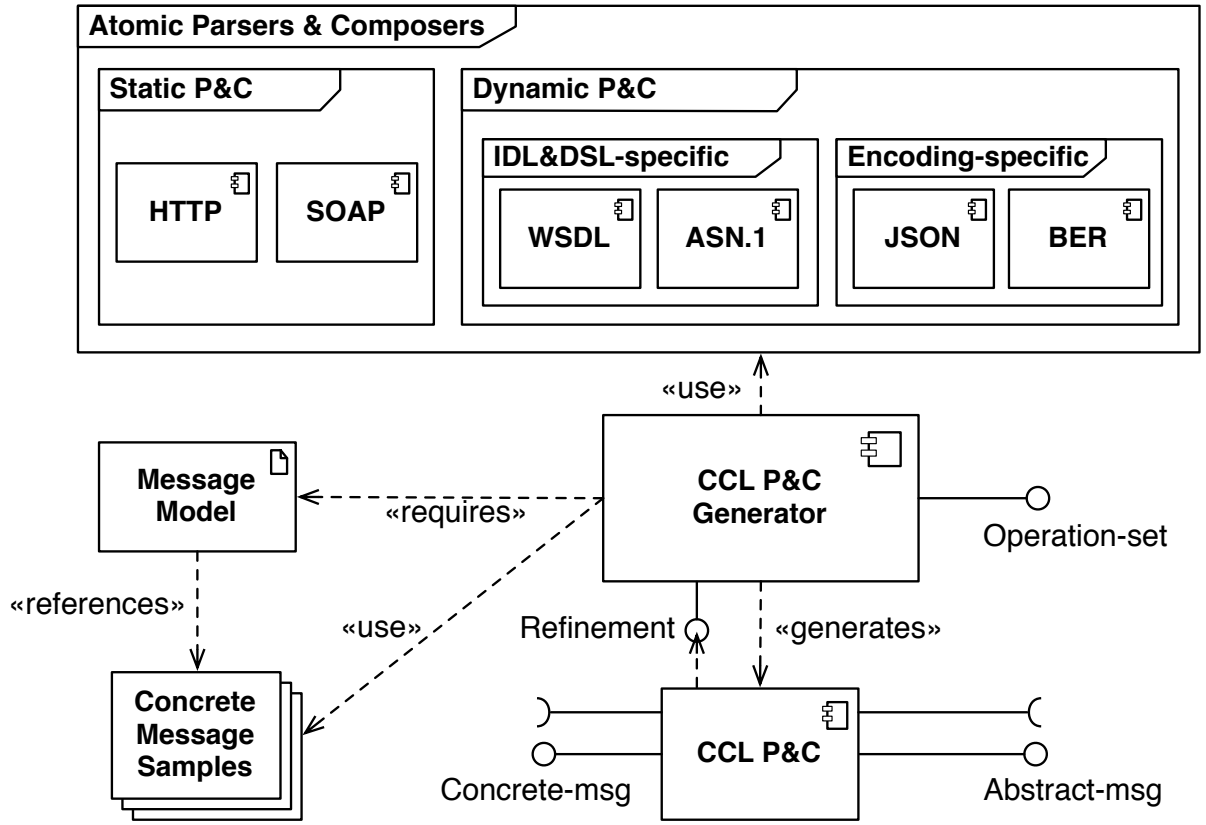

Figure B.2: Message parsing and composition architecture

B.3.1 Cross-Layer P&C Generator Architecture

The goals of the Cross-Layer Parser and Composer Generator (or CCL P&C Generator) are: (i) to make it easier to automatically generate message P&Cs for applications using heterogeneous middleware, (ii) to provide the mediator with an abstract representation of the protocol data that is independent of the specific middleware and application layers used by the interacting components, and (iii) to annotate application data with parsing-related information that is useful to ensure proper mediation. Provided with an abstract representation of application data, mediators can focus on how to enable interoperation between component operations instead of dealing with low level issues such as data encapsulation and communication.

In Fig. B.2, we introduce the architecture used for the CCL P&C Generator. It automatically generates Cross-Layer Parsers and Composers (*CCL P&C*) at design time based on three inputs: (i) a set of **Atomic P&C**s, stored in a repository, that transform a specific data representation format into an uniform parse-tree representation, (ii) a **Message Model** that defines the strategy for assembling *Atomic P&Cs* in order to deal with multiple data encapsulation layers, and, optionally, (ii) a set of **Concrete Message Samples** that allows the engine to refine the generated parser with additional rules discovered from actual messages.

Protocol messages usually contain data scattered over multiple encapsulation layers, each using a different encoding format based on a different standard. Regonline, for instance, uses SOAP over HTTP and SOAP data is encoded in XML. Thus, to have access to the overall message data, at least three different parsers and composers are required. To reduce the effort of generating P&Cs for application messages, our approach relies on modular chaining and re-use of legacy P&Cs, increasing the flexibility and reducing the effort to generate composite P&Cs for specific application messages.

B.3.2 Atomic P&C

We define an *Atomic P&C* as a parser/composer for a well-defined data representation format. A *Composite P&C* is a parser/composer that composes multiple Atomic P&Cs to obtain data and compose a complex application message such as those used by the Regonline service. We classify Atomic P&Cs as either *static* (i.e., obtained by re-using an existing P&C implementation) or *dynamic* (i.e., generated at design-time or run-time from a message description).

Static Atomic P&Cs are more suitable for middleware protocols given that they are based on industry-wide standards, with reference implementations widely available, and are unlikely to change frequently. Dynamic Atomic P&Cs are more appropriate for application-specific protocols, where changes in message structure are frequent.

Dynamic Atomic P&Cs are further classified in two categories, depending on the availability of a message description language: *DSL-specific* and *Encoding-specific*. DSL-specific P&Cs are dynamically generated from a syntactical data-schema available in a machine-readable format, and represent/encode data using an extensible serialization or encoding format (e.g., ASN.1(schema)-BER(encoding), XSD(schema)-XML(encoding)). *Encoding-specific* P&Cs are also built using extensible encoding formats but their message syntax is not necessarily bound to a machine-readable data-schema (e.g, JSON, YAML), but, in most cases, to a human-readable documentation. Given that such messages follow a common encoding format, the syntactic rules, defining the structure of the message, can be inferred or learnt based on a set of message samples. For example the tool Trang[6] can infer an XML schema from one or more example documents.

As seen in Fig. B.2, the CCL P&C Generator supports the refinement of generated P&Cs through the analysis of actual application messages. At design time, encoding-specific message samples can be either synthesized or collected using packet analyzer/capture software and then supplied to the CCL P&C Generator to learn the message structure. The CCL P&C Generator uses CCL P&C instances to partially parse sample messages and produce the necessary input for Atomic P&Cs learning tools, which provide the CCL P&C Generator with feedback to adapt and extend the generated P&C to cope, for example, with new message fields, fields that no longer exist, or extra encapsulation layers.

B.3.3 Composing Atomic P&Cs

In Fig. B.3 we present a fragment of the Message Model for Regonline. For each CCL P&C that is generated in relation with an `operation` (i.e., a pair of request and response messages and associated data) of the component's interface, the Message Model defines, using the `extensions` tag, which Atomic P&C must be used and which part of the message they must process. The `operation` tag specifies the exact operations to which the Message Model applies.

Both `static` parser definitions and `dynamic` parser definitions contain a `path` attribute. This attribute identifies the part of the message the P&C must process. For convenience, the syntax is borrowed from XPath[7]. Extension definitions may also contain an optional attribute `oper`. This attribute defines the operation for which this rule is relevant in the form `Operation:[Request|Response]`. Wildcards can be used on both sides of the attribute to specify that this rule applies to multiple operations or to both requests and responses. Extension definitions also contain an element `identifier`, which defines the Atomic P&C to use. Dynamic P&C definitions require an extra `description` element containing an URI pointing to the message description that must be used to dynamically generate the Atomic P&C and, optionally, a `content` attribute that specifies which part of the message description must be used to process the message element defined by `path`, in the case where the message description contains multiple operations.

A CCL parse-tree instance of the Regonline `GetEvents Request` message is given in Fig. B.4. In this particular case, the definition of the `HttpRequest`'s `body` element contains reference to the `SoapMessage` P&C, which further contains a dynamically generated `WsdlMessage` P&C. The Message Model defines that the initial message element must be parsed with a `HttpRequest` parser, that the `body` element contains a SOAP message that must be parsed with a SoapMessage parser, and finally that the SOAP `body` element must be parsed through a WSDL-defined parser generated dynamically.

B.3.4 Annotating Parsed Data

Cross-layer data dependencies between message encapsulations are a result of complex interactions among protocol layers. As a result, software components may include application data into middleware messages, or add middleware data to the stack of another middleware, further complicating component interoperation. To provide mediators with an abstract representation of application messages, parsed data must be annotated so that they can be transferred to mediators even if they are mixed with middleware data. Moreover, since mediators in our framework handle application-layer mediation, data mapping among different middleware must be handled by the generated CCL P&Cs. Finally, application data can be semantically and syntactically annotated to help mediators find relevant matches between available data and data required to perform an operation.

In our scenario, for instance, Amiando uses a static key called `ApiKey` to control service access while Regonline uses a session id called `ApiToken`. Both data are semantically equivalent, but the mediator should never map the `ApiToken` to `ApiKey` or vice-versa: Amiando will not recognize session keys created by Regonline and Regonline will not accept access keys generated by Amiando. Still, the mediator must have access to this data to map the `ApiToken`

[6] http://www.thaiopensource.com/relaxng/trang.html
[7] http://www.w3.org/TR/xpath/

```
<application name="RegOnline">                                                    1
  <operation>Login</operation><operation>GetEvents</operation>                    2
  <extension>                                                                     3
    <static path="/" oper="*:Request">                                            4
      <identifier>mediation.http.HttpRequest</identifier>                         5
    </static>                                                                     6
    <static path="/body">                                                         7
      <identifier>mediation.soap.SoapMessage</identifier>                         8
    </static>                                                                     9
    <dynamic path="/body/body" oper="GetEvents:Request">                          10
      <identifier>mediation.dynamic.wsdl.WsdlDefinedMessage                       11
      </identifier>                                                               12
      <description>https://www.regonline.com/api/default.asmx?wsdl               13
      </description>                                                              14
      <content>GetEvents</content>                                                15
    </dynamic> ...                                                                16
  </extension>                                                                    17
  <rules>                                                                         18
    <valuerestrict path="/head/uri" oper="*:Request">                             19
      <enumeration value="/api/default.asmx"/>                                    20
    </valuerestrict>                                                              21
    <extract path="/head/headers[name=APIToken]/value" oper="GetEvents:Request">  22
      <fielddef>apiToken</fielddef>                                               23
    </extract>                                                                    24
    <noderestrict path="/head/apiToken" oper="GetEvents:Request">                 25
      <replayonly>true</replayonly><appscope>true</appscope>                      26
    </noderestrict>                                                               27
    <map path="/head/soapAction" oper="GetEvents:Request">                        28
      <source>/body/body/soapAction</source>                                      29
    </map> ...                                                                    30
  </rules>                                                                        31
  <concepts>                                                                      32
    <annotation path="/head/apiToken" oper="GetEvents:Request">                   33
      <concept>SecurityToken</concept>                                            34
    </annotation> ...                                                             35
  </concepts>                                                                     36
</application>                                                                    37
```

Figure B.3: Fragment of the Message Model for the Regonline component

between the Regonline `LoginResponse` message and the Regonline `GetEventsRequest`. We call this type of field **replay-only** because it must be mapped only within the set of operations from the same component. Another example was mentioned earlier: after performing a `Login` operation, Regonline returns the `ApiToken` (application layer). All further interaction requires this value to be passed as an HTTP Header (middleware layer). The generated P&C must know that, even though the `ApiToken` is found at a middleware encapsulation layer, its **scope** is the application layer. Finally, the presence of some message fields may be **optional**. In our scenario, the Regonline `GetEvents` operation accepts an optional `orderBy` parameter to specify the return order of conferences. If the mediator is unaware that this field is optional, it may fail to map an operation between components because a required input is not provided.

The `rules` section of the Message Model supports the definition of such syntactic rules concerning parsed data (see Fig. B.3). The `noderestrict` tag defines field level conditions on data. The `replayonly` element marks this data as available for mapping only among operations from the same component. The element `appscope` marks that the P&C must consider this data as part of the application scope to solve cross-layer encapsulation issues. Finally, the `optional` element defines the annotated data as not required for executing the operation. Another element of the `rules` section is `valuerestrict`, which allows specifying detailed value patterns for simple data types. We borrow this fragment of the language from XSD restrictions. While it may increase the complexity of the specification, this feature leads to a more precise data-mediation and message-validation than relying only on type-definition and semantical annotations. The `extract` element allows selecting specific fields out of a sequence, and attach them as an identifiable field at the same level in the parse-tree. Finally, the `map` element enables to hardwire middleware layer mappings

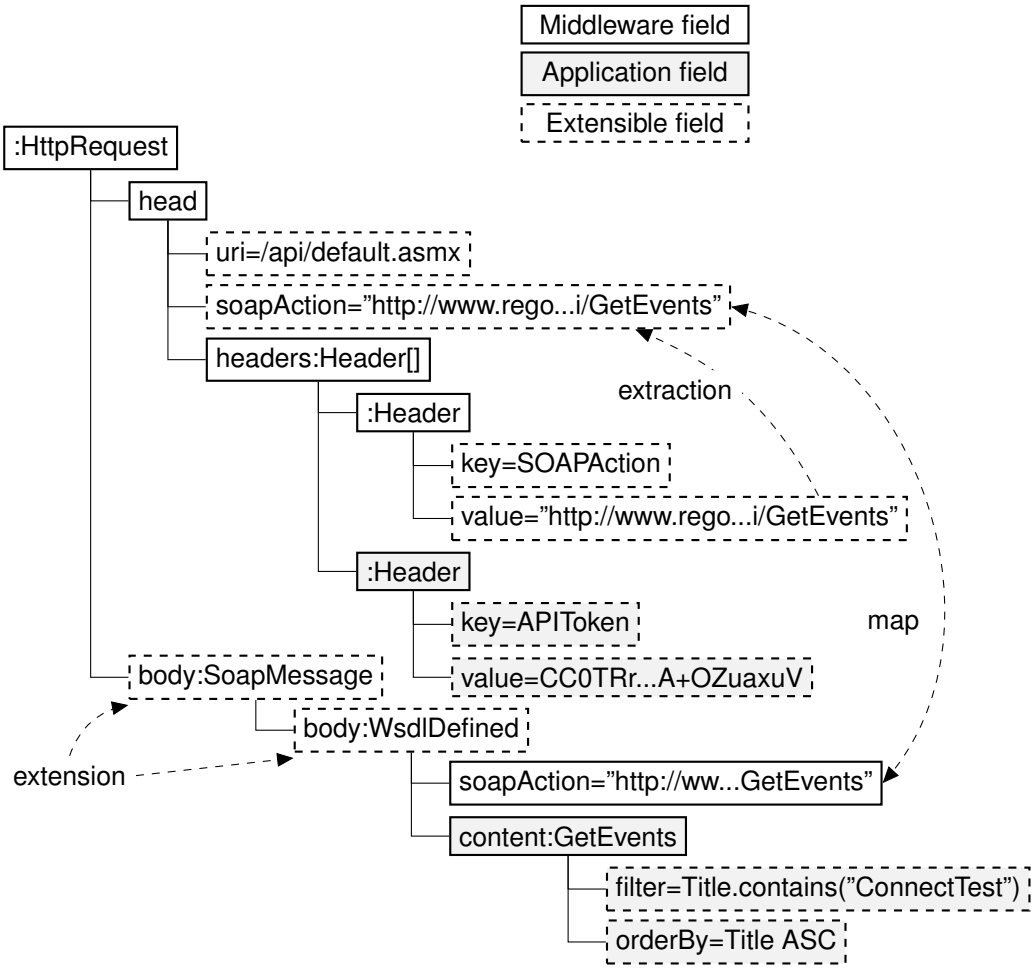

Figure B.4: Parse-tree instance of Regonline GetEvents-Request message. Arrows point to fields where the Message Model has effect.

into the CCL P&C. Since the mediator is completely unaware of the middleware layers and the differences between each component's middleware, generated CCL P&C must handle middleware interoperability and map middleware data between different middleware layers. For example, in the case of the message instance illustrated in Fig. B.4, and described in the Message Model fragment in Fig. B.3, the WSDL P&C meta-information field `body/body/soapAction` is mapped to the HTTP request header field `/head/soapAction`.

The last section of the Message Model, `concepts`, enables the semantic annotation of parsed data, with variable granularity. For example, one may annotate an operation, a message, and/or any message field (either of complex or simple type).

Besides the generation of the P&C, the analysis of the Message Model associated with a component results in a uniform description of the component's operations, from which corresponding request and reply messages directly derive. This description is taken as input by the mediator synthesis process that is discussed next.

B.4 Abstract Mediators

To synthesize an abstract mediator, we focus on inferring the transformations needed to make data flow between the two components and enable them to interact successfully. Note however that we are not able to generate an abstract mediator if a component requires an operation that cannot be provided by the other. Indeed, the mediator can aggregate, dispatch, transform, cache and forward data but it cannot create data, i.e., provide an operation by itself.

B.4.1 Supporting Reasoning Mechanisms

Essential to enable the automated reasoning about the interoperation of heterogeneous systems and the synthesis of the associated mediators is the formal modeling of the semantics of these systems as well as the related domain.

Ontologies to reason about domains

ontologies provide a powerful formalism to represent and reason about domain knowledge by making explicit the concepts and properties of this domain through a set of logical axioms [6]. Usage of domain-specific information encoded within an ontology is at the core of the mediation-based approach to information integration [115] and data integration across databases [21]. Still, to the best of our knowledge, our work is the first that uses ontologies to support component interoperability from application down to middleware. In particular, while analyzing the Message Model, the CCL P&C generator provides a uniform description of the operations provided and required by the component in terms of request and response messages. Each message definition contains references to the domain ontology concepts that describe the semantics of the data it encloses, as expressed by the `concept` tag in the associated Message Model. As a result, we can abstract the interface of a component through *required* and *provided actions*.

Let denote a required action that represents a client-side invocation of an operation op, as $\alpha =< op, I, O >$ where op, I, and O are concepts in a given ontology. The given invocation is specifically performed by sending a request containing data I and receiving the corresponding response that include data O. A provided action, written $\overline{\beta} =< \overline{op}, I, O >$, represents a server supplied operation, which is performed by receiving a request message and sending the corresponding response message.

For the synthesis of mediators, we are interested in checking specialization/generalization between concepts, that is the *subsumption* relation. Ontology-based subsumption resembles in many ways to type subsumption [20], that is if a concept C is subsumed by D, written $C \sqsubseteq D$, then all instances of C are also instances of D and all the properties of D are also applicable to C. Ontology-based subsumption further considers composite concepts such as concepts constructed using the disjunction operator (written $\sqcup$), which defines a concept as the union of the instances of two (or more) concepts. Hence, our approach is ontology-aware in the sense that it relies on existing ontologies and uses ontology reasoners to verify subsumption between concepts that might be atomic or composite.

LTS to reason about component behavior

besides the functional semantics of components defined using references to a domain ontology, it is important to describe how the actions of the interface are coordinated in order to achieve the component functionality, that is the behavioral semantics of systems. We use Labeled Transition Systems (LTS) to describe the behavioral semantics of components. LTS provide an abstract and wide-spread, yet precise, formalism to represent and analyze system behavior. The LTS of a component is a quintuplet $P = \langle S, \mathcal{I}, \rightarrow, s_0, F \rangle$ where:

- S is a finite set of states,
- $\mathcal{I}$ is the interface of the component and represents the alphabet of the LTS,
- $\rightarrow \subseteq S \times \mathcal{I} \times S$ denotes a transition relation that specifies that if the component is in state $s \in S$ and engages in an action $a \in \mathcal{I}$, then it transits to state $s' \in S$, written $s \xrightarrow{a} s'$. $s \overset{X}{\Rightarrow} s', X = \langle a_1, ..., a_n \rangle, a_i \in \mathcal{I}$ is a shorthand for $s \xrightarrow{a_1} s_1 ... \xrightarrow{a_n} s'$.
- $s_0 \in S$ indicates the initial state, and
- $F \subseteq S$ denotes the set of final states.

We can reasonably assume that the LTS describing the behavior of a component is either provided with or derivable from the component interface. On the one hand, there are various approaches and standards that emphasize the need and the importance of having such a complete specification. On the other hand, there are mature learning techniques and tools to support the automated inference of LTS models [12].

Based on the description of two components using their interfaces $\mathcal{I}_1$ and $\mathcal{I}_2$ and their behaviors P_1 and P_2 respectively, we first map their interfaces such that the actions required by one component can be provided by a sequence of actions of the other components. Then, we analyze their behaviors so as to compose the obtained mappings such that both of them evolve synchronously until reaching their final state. This composition is the mediator.

B.4.2 Mapping Interfaces

A sequence actions required by component $\left\langle \alpha_i = \langle a_i, I_{a_i}, O_{a_i} \rangle_{i=1..m} \in \mathcal{I}_1 \right\rangle$ can be achieved using a sequence of actions provided by the other component $\left\langle \overline{\beta_i} = \left\langle \overline{b_i}, I_{b_i}, O_{b_i} \right\rangle_{j=1..n} \in \mathcal{I}_2 \right\rangle$ only if some constraints are verified. The first is that the provided operation are a specialization of the required ones, that is $\bigsqcup_{j=1}^{n} b_j \sqsubseteq \bigsqcup_{i=1}^{m} a_i$. When a component requires an action, it sends the input data if this action and receives the corresponding output data. Hence, we can allow the component to progress if the input data are cached and it does not expect any output data. Let us set l as the position of the first required action that necessitates some output data, i.e., $\forall h \in [1, l[,\ O_{a_h} = \emptyset$. To provide this output data, the provided actions have to be executed. Therefore, the input data of the first action can

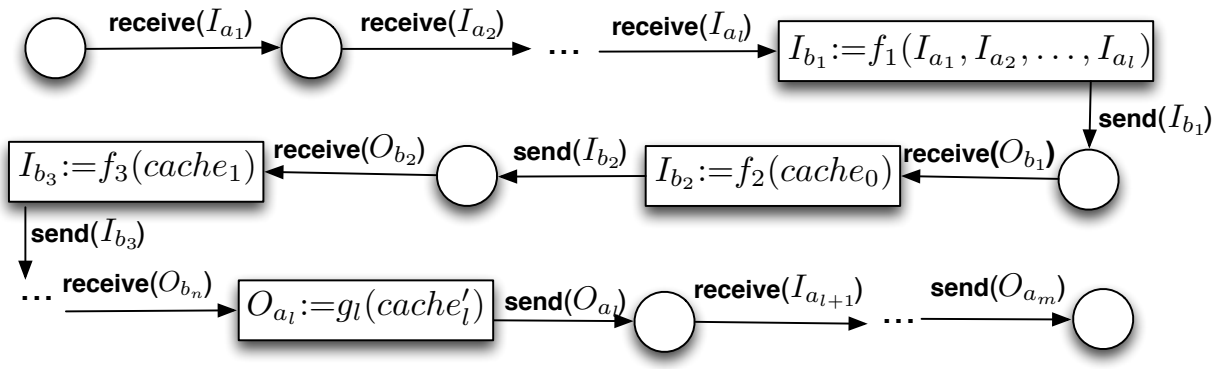

Figure B.5: Many-to-many mapping

be obtained using the received input data, i.e., $\bigsqcup_{i=1}^{l} I_{a_i} \sqsubseteq I_{b_1}$. Once this action has been executed, then the cached data are augmented with its output data. In general, when $i-1$ provided actions has been performed the cache is $cache_{i-1} = \left(\bigsqcup_{j=1}^{l} I_{a_j}\right) \bigsqcup \left(\bigsqcup_{h=1}^{i-1} O_{b_h}\right)$. Then, the input of the i^{th} action is obtained using the cached data, i.e., $cache_{i-1} \sqsubseteq I_{b_i}$. Once all the provided actions have been performed, we can execute the remaining required actions if all their output are available. i.e., $\forall h \in [l,m], cache'_h \sqsubseteq O_{a_h}$ where the cache is augmented with the input data of the required actions, that is $cache'_h = \left(\bigsqcup_{i=l}^{h} I_{a_i}\right) \bigsqcup cache_n$. We use constraint programming to compute the mapping efficiently. The conditions on the operations and input/output data represents the constraints. The solver implements intelligent search algorithms such as backtracking and branch and bound to optimize the time for finding the mapping.

Subsumption guarantees that the semantics of data are preserved. Still, we need to specify the syntactic transformations necessary in order to convert data received from one component to that expected by the other. When dealing with simple types, the function consists in an assignment but when dealing with complex types we must compute the translation needed to transform one schema into another. To that end, we rely on existing schema matching techniques [103]. In the most general case, schema matching is very complex and can hardly be fully automated. However, in our case, we just match specific data elements knowing that they already match semantically. For example, the output of the `GetEvents` operation retuned by the Regonline service includes an `Event` data and the Amiando Client also expects an `Event` data to be returned by the `EventRead` operation. Both are annotated using the Event concept even though their XML schema are different. We use the Harmony library [88] to find the matching pairs of elements between this two schemas. Harmony combines different matching techniques (e.g., bag of words, thesaurus, and WordNet) to establish the correspondence between the elements of the source and target schemas with the best confidence score possible. The translation functions consist in assigning the matching pairs with the higher confidence score. Then we assign default values to the remaining elements if they accept one. Otherwise, the mapping is not valid even though the actions are semantically compatible.

In Fig. B.5 we represent a many-to-many mapping process, written $Map(\langle \alpha_1, ..., \alpha_l, ..., \alpha_m \rangle, \langle \overline{\beta_1}, ..., \overline{\beta_n} \rangle)$. The $l-1$ first required operations are performed since they do not require any output. To execute β_1, we need to transform the cached input data into the input expected by β_1 using the input translation function f_1. In general, input data translation is performed by f_i functions while the output data are translated by the g_j functions until the mapping is completed.

B.4.3 Synthesizing Abstract Mediators

While the interface mapping ensures that a sequence of actions from one component can safely be achieved using another sequence of actions from the other components, it does not specify when each mapping has to be performed. Furthermore, the interface mapping might be ambiguous, i.e., the same sequence of actions may be mapped to different sequences of actions. Hence, the mappings need to be combined such that the interaction of the two components does not lead to erroneous states, e.g., a deadlock. We analyze the behavioral specifications of the two components, which are specified using the LTSs P_1 and P_2 respectively, and generate a third LTS, the mediator M, such that the mediated components reach their final states, which implies that the system made up of the parallel composition of P_1, P_2, and M is free from deadlock, or we determine that no such mediator exists. If a mediator exists, then we say that P_1 and P_2 can interoperate using a mediator M, written $P_1 \leftrightarrow_M P_2$.

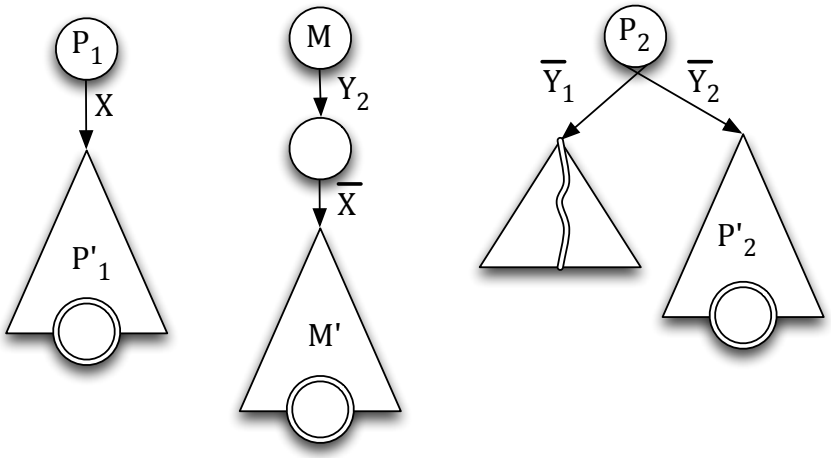

Figure B.6: Incremental Synthesis of a Mediator

We inductively build a mediator M by forcing the two components to progress consistently so that if one requires the sequence of actions X, the interacting component must provide a semantically-compatible sequence of actions $\overline{Y}$. Given that an interface mapping guarantees the semantic compatibility between the actions of the two components, then the mediator is able to compensate for the differences between their actions by performing the necessary transformations.

The base case is that where both P_1 and P_2 are final states. In which case the mediator is made up of one final state. Then, at each state we choose an eligible mapping, i.e., both processes can engage in the sequences specified by this mapping and moves to states where a (sub) mediator can be generated, which is formally described as follows.

if $P_i \stackrel{X}{\Rightarrow} P'_{3-i}$ and $\exists(X, \overline{Y}) \in Map\,(\mathcal{I}_i, \mathcal{I}_{3-i})$

such that $P_{3-i} \stackrel{\overline{Y}}{\Rightarrow} P'_{3-i}$ and $P'_i \leftrightarrow_{M'} P'_{3-i}$

then $P_i \leftrightarrow_M P_{3-i}$ where $M = Map(X, \overline{Y}); M'$

This implies that when multiple mappings can be applied, we select one of them and check if it allows the two processes (P_i and P_{3-i}) to reach their final states. Otherwise, we backtrack and select another mapping. Let us consider the example in Fig. B.6. P_1 is ready to engage in X, P_2 can execute either $\overline{Y_1}$ or $\overline{Y_2}$, and X can be mapped to both $\overline{Y_1}$ and $\overline{Y_2}$. If we select the first mapping, the final state of P_2 cannot be reached. Hence, we backtrack and select the second one. To compute the mediator M, we append the mapping of $Map(X, \overline{Y_2})$ to M', which is the mediator computed between P'_1 and P'_2. If none of the mapping is applicable, then we are not able to generate the mediator. Note however, that the mediator is not unique since many mappings may lead P_i and P_{3-i} to their final states. In this case, we only keep the first valid mapping.

B.5 Implementation and Validation

In order to demonstrate the practicality of the proposed mediation framework we have implemented a standards-based prototype using Java. We validate our approach by achieving interoperability between the RegOnline and Amiando conference management systems and comparing results with the baseline, non-mediated, systems. On the server-side we use the services provided by Amiando and Regonline. On the client-side we use a Java implementation provided by Amiando, while for Regonline, we generate the client using the `wsimport`[8] tool and the WSDL service description. We compare the mediated execution-time with the non-mediated case. Each test was repeated 30 times, in similar conditions, and connection delays were excluded (e.g., opening sockets, SSL handshake, etc).

We generate two mediators: one for mediating the Amiando Client in order to connect to the RegOnline service, and a reverse mediator for the RegOnline client to connect to the Amiando service. To generate the mediators, we first provide a Message Model[9] for each system, as well as two message samples (obtained using a network packet analyzer) containing the JSON formatted responses of the Amiando service. The CCL P&C Generator is then able

[8] http://docs.oracle.com/javase/6/docs/technotes/tools/share/wsimport.html

[9] https://www.rocq.inria.fr/arles/software/ccl-Mediation-Framework/

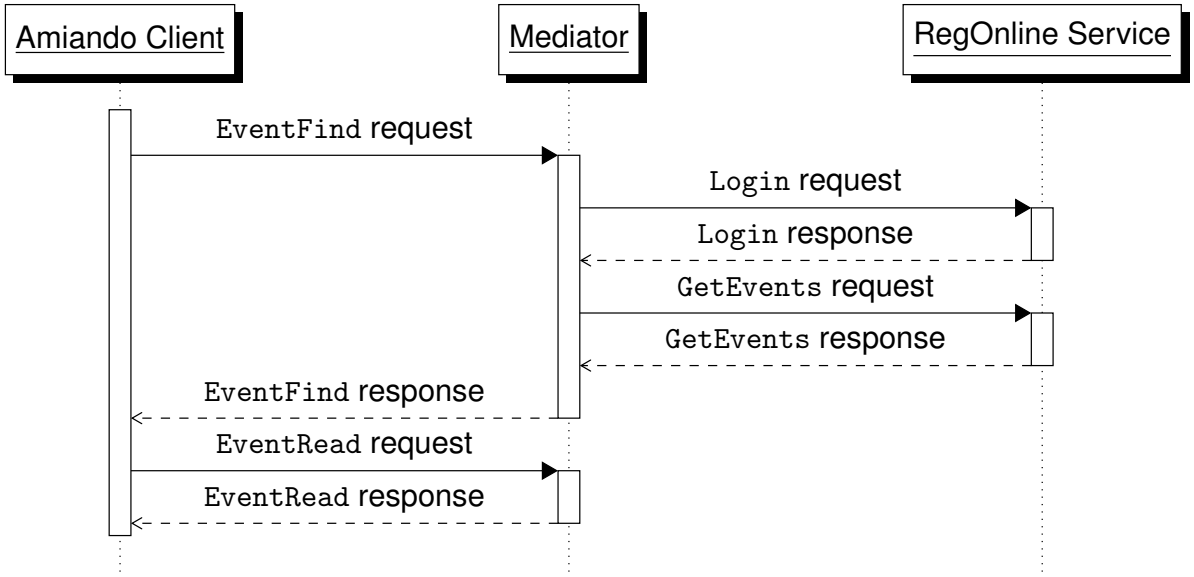

Figure B.7: Case study interoperability scenario

to generate eight different CCL P&Cs and their associated SAXSD[10]. The SAXSD is obtained by injecting semantical annotations defined in the Message Model into the XSD document generated using JAXB[11]. Based on the SAXSD, the provided domain ontology and the LTS behavioral description, the Abstract Mediator Synthesis component generates a mediator. The mediator and associated CCL P&Cs are executed by the Mediator Engine, following the sequence of operations presented in Fig. B.7. For generating the reverse mediator we did not have to provide any additional inputs.

In Fig. B.8, we evaluate the execution-time overhead of the mediation. Since this test is performed using the real online services, the response time varies depending on the network conditions. As expected, the mediated execution-time is superior to the non-mediated case, given that the number of messages exchanged is doubled. We show the decomposition of the execution-time for mediation, composing and access/parsing. Network access and parsing cannot be distinguished in this case because parsing is done in multiple steps when data is available on the communication channel. While the overhead of mediation and message composition is low, we see that parsing and network reception introduce the largest overhead. This is why, in Fig. B.9, we detail the decomposition of parsing time over each Atomic parser chained in the generated CCL P&Cs. We see that the `EventFind` response message parsing has a peak of 1662 ms. We also observe that the entire time is associated with the HTTP parser, and given that the size of the message is only 869 bytes, we can conclude it is almost entirely due to the response delay of the Amiando Service. The same reasoning applies for the `GetEvents` response message of the RegOnline service, but in this case 197 ms are associated with the SOAP parser which is chained to parse the HTTP response's payload (the HTTP body). Knowing that in this particular implementation, the SOAP parser does not wait for network access, we observe that the SOAP Atomic parser introduces an important SOAP-Envelope parsing overhead. This observation confirms that the Amiando/RegOnline mediator execution-time (in Fig. B.8) can be reduced by using a more efficient SOAP Atomic parser.

Comparing to the non-mediated tests, we can conclude that our mediation approach introduces an acceptable overhead while enabling seamless interoperability between the two systems.

B.6 Related Work

In [61], we survey existing approaches to mediation and give initial thoughts about leveraging ontologies to deal with interoperability from application down to middleware. In this paper we give a detailed approach on how to actually achieve the mediation and evaluate it with a real-world scenario. In this approach, ontologies are only used for mediation at the application level, while libraries and learning techniques are used to perform the necessary parsing and composing of middleware-specific messages. In this section, we discuss two categories of related work: network protocol message processing and automated mediation.

[10] Semantically annotated XSD. http://www.w3.org/2002/ws/sawsdl/
[11] Java Architecture for XML Binding. http://jaxb.java.net/

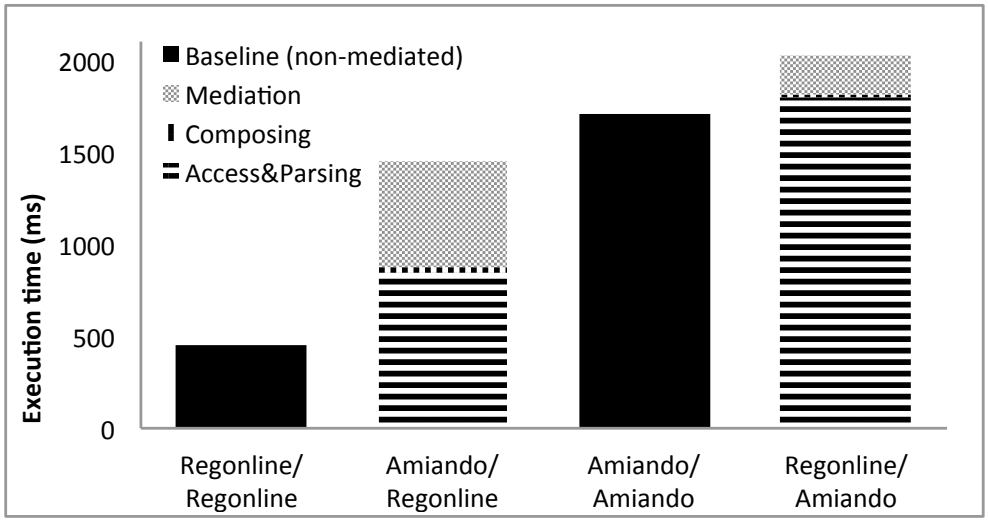

Figure B.8: Comparison between mediated and non-mediated executions. For the mediated case we provide the approximate time required for Parsing, Composing and Mediation.

Network protocol message processing

Most approaches to processing protocol messages [80, 23, 29] rely on a specification of the structure of messages using variations of the ABNF metalanguage. While these approaches are highly efficient, they lack flexibility since they require a very low-level specification of the exchanged messages each time a parser or composer must be generated and the whole specification must be re-written if one of the protocols from the stack changes. Instead of considering protocols as monolithic blocks, Model Driven Architecture (MDA)[12] proposes to specify applications using an abstract model, called the Process Independent Model (PIM). The PIM is deployed atop middleware platforms described by the Platform Specific Model (PSM). This decoupling enables the modeling of application-middleware data dependencies, which improves traceability and facilitates consistency management in the development process. However, MDA does not specify how to deal with heterogeneous PSM or PIM models. Starlink [27] proposes a framework to bind application-level mediators to different middleware based on a DSL specification of both components as well as the mediator. However, the binding does not support a hierarchy of protocol layers and has to be specified per application. For example, to define SOAP over HTTP binding, we cannot reuse the HTTP binding but we have to specify a new binding instead. Finally, the mediator needs to be specified manually by the developer who must know both systems and ensures that the translation is correct.

Automated mediation

Mediation has long been advocated as a practical means to compose heterogeneous systems in various domains, e.g., database integration [21], component adaptation [119], middleware bridging [27], connector wrapping [106], and Semantic Web Services composition [44]. While mediation was mainly a manual task, the increasing complexity, heterogeneity, and openness of today's complex systems makes the automated synthesis of mediators a major concern. Semi-automated approaches [84, 91] focus on exploring the behavior of the components to be mediated assuming a partial specification of the mapping of their operations and data. Full automation can be reached either by measuring the similarity between data types using statistical methods [91] or by relying on ontologies to verify their semantic compatibilities. The latter approach is more accurate since it considers logical axioms rather than syntactic distance. In this sense, the work of Cavallaro *et al.* [33] is the closest to ours. They consider the semantics of data and rely on model checking to identify mapping scripts between interaction protocols automatically. Nevertheless, they are not able to manage ambiguous interface mapping, i.e.,when the same action may be mapped to different actions since they align the actions of the two components *a priori*. Hence, solutions to mediator synthesis consider one abstraction only: application and assume the use of SOAP underneath. Consequently, they fail whenever it is necessary to deploy the mediator atop different middleware.

[12] http://www.omg.org/mda/

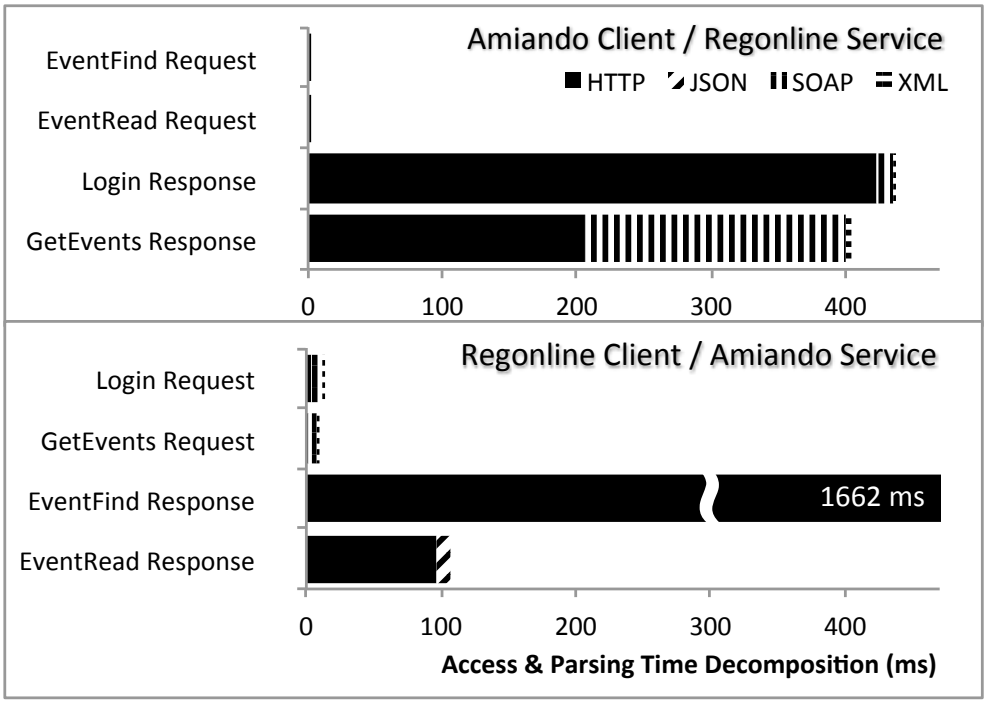

Figure B.9: Parsing overhead decomposed by Atomic parsers

B.7 Conclusion

In this paper we have presented an automated mediation framework to enable protocol interoperability from application to middleware layers. We have shown that this framework successfully enables heterogeneous conference management systems to interoperate in a transparent way introducing acceptable overhead, which remains within the bounds of expected performance from the legacy applications. Future work includes increasing automation by inferring, at least partially, the Message Model by cooperating with discovery mechanisms and packet-inspection software, and experimenting also extracting the inference of component behavior with various learning techniques. Finally, incremental re-synthesis of mediators and, run-time refinement of CCL P&Cs would be useful in order to respond efficiently to changes in the individual systems or in the ontology. A further direction is to consider improved modeling capabilities that take into account the probabilistic nature of systems and the uncertainties in the ontology. This would facilitate the construction of mediators where we have only partial knowledge about the system.

C Synthesizing Connectors meeting Functional and Performance Concerns

Antinisca Di Marco (UNIVAQ), Paola Inverardi (UNIVAQ), and Romina Spalazzese (UNIVAQ)

Abstract. Today's networked environment is characterized by a wide variety of heterogeneous systems that dynamically appear with the aim to interoperate to achieve some goal. In this evolving context, there is no a-priori knowledge of the systems and automated solutions appear to be the only way to achieve interoperability with the needed level of flexibility. We already proposed an approach to the automated synthesis of CONNECTors (or mediators) to reconcile protocols diversities and to allow systems interoperability.

In this paper we enrich our approach to synthesize CONNECTors to take into account, together with functional concerns, also performance aspects. By reasoning on systems' specification, the first step of the approach we are proposing produces a mediator that satisfies the functional requirements. The second step, by considering specific strategies, acts on the produced mediator to satisfy also the performance ones.

Keywords: Connector synthesis, Interoperability, Performance

C.1 Introduction

Nowadays the networked environment is more and more characterized by a wide variety of heterogeneous systems that dynamically appear and concur to achieve some goal through interoperation with other systems. Systems' goal can be about functional and/or non functional aspects that have to be satisfied in order for two systems to interoperate.

Abstractly some of these heterogeneous applications could interact, since they have compatible interaction protocols. Nevertheless they can be characterized by some resolvable mismatch in their protocols (e.g., order interactions or input data formats) and non functional requirements that may undermine their ability to seamlessly interoperate. Solving such mismatches and meeting the non functional requirements, asks for applications' adaptation through a /connector. Further, in this evolving context there is no a-priori knowledge of the systems and automated solutions appear to be the only way to enable composition and interoperability of applications with the needed level of flexibility. The described context is considered by the CONNECT European project[1] whose aim is to allow seamless interoperability between heterogeneous protocols at various levels. The project adopted as solution an approach for the on the fly synthesis of *emergent* CONNECTors via which Networked Systems (NSs) communicate. The emergent CONNECTors (or mediators) are system entities synthesized according to the behavioral semantics of protocols run by the interacting parties at application and middleware layers. The synthesis process is based on a formal foundation for CONNECTors, which allows learning, reasoning about and adapting the interaction behavior of NSs at run-time through CONNECTors. Some of the authors proposed the approach to the automated synthesis of CONNECTors to reconcile protocol diversities from a functional point of view that suits the CONNECT vision [59, 60].

In this paper, we enhance the approach to synthesize CONNECTors to take into account, together with functional concerns, also performance aspects. By reasoning on systems' specification, the first step of the approach produces a mediator that satisfies the functional requirements only. The second step takes into account specific strategies that improve the system performance to target performance requirements -if possible.

The original contributions of this paper are: *i)* the definition of the process combining the CONNECTor synthesis and the analysis-based reasoning on it to meet non functional requirements on the whole connected system (that is the one composed by the NSs plus the synthesized mediator), *ii)* the identification of three general strategies useful to improve the system performance, and *iii)* the introduction in the synthesis process of a performance analysis-based reasoning step aiming at refining the synthesized CONNECTor to meet the performance constraints.

The reminder of the paper is organized as follows. Section C sets the context also recalling background notations for the NSs specification. Section C introduces a case study. Section C gives the overview of our enhanced approach. Section C details the performance analysis-based reasoning step and reports assessments about the effectiveness of our approach. Section C discusses related works and finally Section D concludes the paper.

C.2 Setting the Context

Figure C.1 gives an overview of the context we consider. A number of heterogeneous networked systems, e.g., Tablet, Server, Smartphone and Desktop are dynamically available in the networked environment. NSs heterogeneity spans

[1] CONNECT Web Site - http://connect-forever.eu/

many aspects and we focus on the application layer heterogeneity. An example of the heterogeneous applications we consider is represented in Figure C.1 and will be detailed in Section C. For instance, Smartphone in Figure C.1 has a *shopping client* application represented by the shopping cart icon. Its shopping client is compatible/complementary with the Server *shopping server* application represented by the seller icon. However, due to some protocol discrepancies and non functional concerns, they cannot seamlessly interoperate and a CONNECTor that mediates their differences is needed in between.

In this paper, and in CONNECT too, we consider that the **NSs are black box** and that they expose in the interface: their interaction behavior description, their non functional properties description, and possibly their non functional requirements on potential interactions with others NSs. Thus, **the CONNECTor is the only locus where we can act to make compatible the NSs behaviors**. From a functional point of view, it can only interact with NSs interaction behavior. Hence, for instance, if a NS sends data with a finer granularity with respect to another NS, the mediator has to first collect all the pieces of data information and then properly send them to the other. In the described scenario, performance concerns arise because a NS wants to achieve a goal, that requires interaction with other NSs, with some performance constraints. It can happen that the synthesized mediator satisfies the functional concerns but, when assembled with the NSs, the connected system does not satisfy the performance requirements. To overcome this problem, we can act on the mediator to satisfy the non functional requirement. We identify three general *strategies* that can be applied singly or in combination: i) *alternative* CONNECTor *behaviors slicing* that can be applied if at least one of the NSs has alternative protocol behaviors. In this case the CONNECTor behavior is sliced and it mediates only a subset of the NSs alternatives; ii) *tuning the upperbound number of loop iterations*, several bounds are considered in the analysis and only the ones that help in satisfying the performance requirements are considered in the final synthesized CONNECTor. Finally iii) *deployment configuration* that highlights the most convenient deployment among three possibilities: *all remote components* where the mediator and the NSs are deployed on separate machines, and *local to NS1* or *local to NS2* where the mediator is deployed on the same machine where NS1 or NS2 is running, respectively.

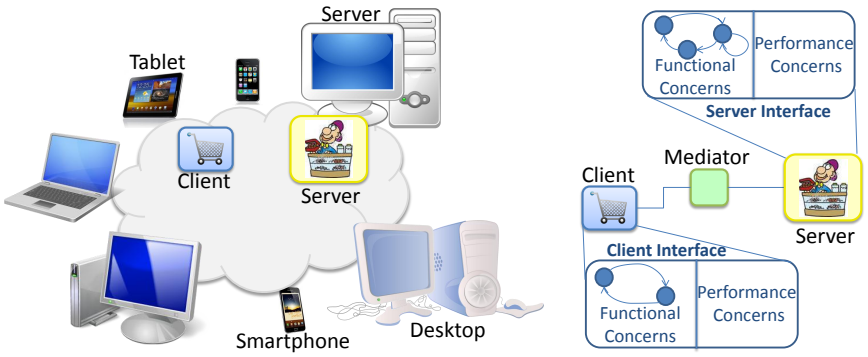

Figure C.1: An overview of the context

Background In the following, we recall notations inherited by the CONNECT project to specify NSs' protocols and non functional quantitative concerns.

Networked Systems' protocol specification. We use Labeled Transition System (LTS) to model systems' protocol and refer to ontologies to conceptualize systems' actions and input/output data, and to reason on them.

Specifically, we consider what we call *enhanced Labeled Transition Systems (eLTS)* that is a quintuple (S, L, D, F, s_0) where: S is a finite non-empty set of states; L is a finite set of labels describing actions with data; $D \subseteq S \times L \times S$ is a transition relation; $F \subseteq S$ is the set of final states; $s_0 \in S$ is the initial state. The eLTS' labels are of the form $<op,In,Out>$ where: op is an observable action referring to an ontology concept or is an internal action denoted by τ; an action can have output/input direction denoted by an overbar or no overbar on the action respectively, e.g. $\overline{act}$ or act. In, Out are the sets of input/output data that can be produced/expected whose elements refer to ontology elements. We are able to describe the following actions with data: (1) *output action with outgoing parameters and incoming return data* $<\overline{op}, In, Out>$ where In is produced while Out is expected; (2) *input action with incoming parameters and outgoing return data* $<op, In, Out>$ where In is expected while Out is produced. One at a time In or Out might be empty because no input/output data is expected/produced. This leads to 4 variants of the actions, two for (1) and two for (2). Note that (1) ($<\overline{op},In,Out>$) can be equivalently described by the two following action primitives: $<\overline{op},In,->$ and $<\overline{op},-,Out>$. This applies similarly to (2) ($<op, In, Out>$ can be described as $<op,In,->$ and $<op,-,Out>$). Between protocols, we assume synchronous communications on complementary actions. Actions $<op_1,In_1,Out_1>$ and $<op_2,In_2,Out_2>$ are complementary iff $op_1 = \overline{act}$ and $op_2 = act$ and $In_2 \subseteq In_1$ and $Out_1 \subseteq Out_2$ (or similarly with exchanged roles of op_1 and op_2). Moreover, we consider finite traces by assuming a bound on the number of cycles execution.

Ontologies describe domain-specific knowledge through concepts and relations, e.g. the subsumption: a concept C *is subsumed by* a concept D in a given ontology $\mathcal{O}$, noted by $C \sqsubseteq D$, if in every model of O the set denoted by C is a subset of the set denoted by D [6]. We assume that each NS action and datum refer to some concept of an existing domain ontology so that we can reason on them in order to find a common language between protocols.

Non functional concerns specification. We use Property Meta-Model (PMM) [42] to specify non functional con-

cerns, i.e., the non functional requirements the connected system must satisfy, and the non functional characteristics of the NSs and of the synthesized mediator actions. PMM is a machine-processable specification language for non functional properties and metrics that spans dependability, performance, security and other complex properties (such as trust) that can be defined upon others properties. A property can be of two types: PRESCRIPTIVE that represents properties required for some system, or DESCRIPTIVE that represents non functional properties owned by a system or by one of its actions. Figure C.2(a) shows a response time DESCRIPTIVE property (1 ms) of an action *login* whereas Figure C.2(b) shows a PRESCRIPTIVE property representing a throughput requirement on a connected system.

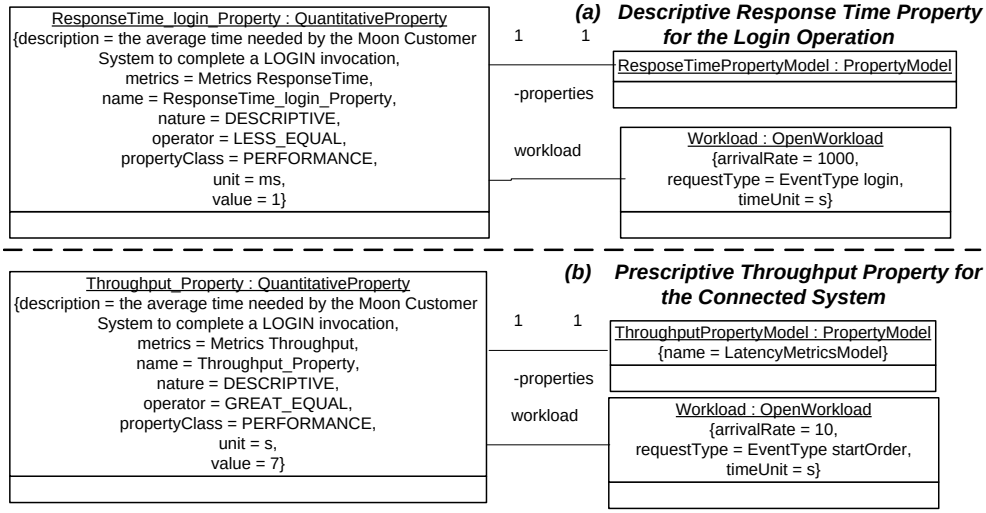

Figure C.2: PMM models for Descriptive and Prescriptive Properties

C.3 Case Study

To better explain our proposal, we introduce the *Blue-Moon* case study within the e-commerce domain. It is an instance of the abstract example described in Section C and is based on the Purchase Order Mediation scenario of the Semantic Web Service Challenge (SWSC)[2]. The scenario describes a typical real-world problem and highlights various mismatches that can happen when making heterogeneous systems interoperable.

One of the systems is a customer ordering products called *Blue* that has two alternative behaviors and whose protocol is illustrated by the eLTS in Figure C.3. The Blue purchase process in the beginning starts an order and then, two alternative behaviors can occur. Either (i) one or many items can be added one at a time to the order, or (ii) the order can be initiated by sending a list of all the items. Then the order can be placed and a confirmation is expected for each item belonging to the order. Based on the kind of order done, the confirmations will be received one at a time or all together. The purchase concludes by receiving the result of the payment.

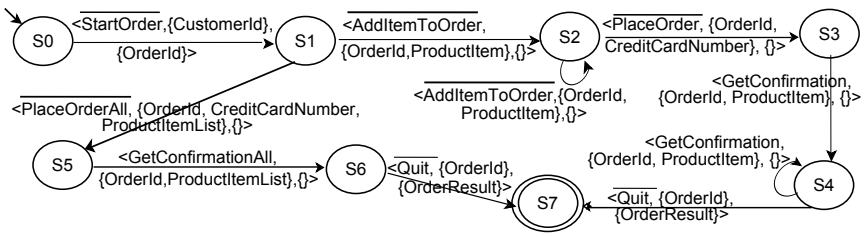

Figure C.3: Blue Client

The other system is called *Moon* and its protocol is illustrated in Figure C.4. First, it provides to authorized customers the information to create an order. Then, Moon can

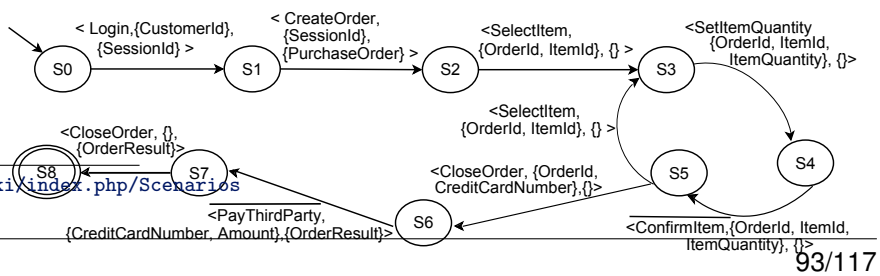

Figure C.4: *Moon* customer service networked system

[2] http://sws-challenge.org/wiki/index.php/Scenarios

perform one or many times the following operations: it receives requests to add individual items to the created order, by receiving the item selection and the respective needed quantity, and it provides the confirmation about the required item. When the order is complete, Moon proceed to perform the payment using a third-party system for its management, and close the order by sending the payment result to the customer.

We assume that Blue comes with the following PRESCRIPTIVE property in its description: the average throughput of the connected system is greater than 7 (where connected system means all the applications that cooperate to purchase an order) when it generates the workload of 10 requests per second (this requirement is specified as PMM model in Figure C.2(b)). Further, from the ontological description of Blue and Moon we can infer the correspondences reported in Table C.1 that will be used to build the mediator[59].

$\overline{StartOrder}, \{CustomerId\}, \{OrderId\}$	$Login, \{CustomerId\}, \{SessionId\}$ $CreateOrder, \{SessionId\}, \{PurchaseOrder\}$
$\overline{AddItemToOrder}, \{OrderIdProductItem\}, \{\}$	$SelectItem, \{OrderId, ItemId\}, \{\}$ $SelectItemQuantity, \{OrderId, ItemId, ItemQuantity\}, \{\}$
$\overline{PlaceOrder}, \{OrderId, CreditCardNumber\}, \{\}$	$CloseOrder, \{OrderId, CreditCardNumber\}, \{\}$
$\overline{PlaceOrderAll}, \{OrderId, CreditCardNumber, ProductItemList\}, \{\}$	$(SelectItem, \{OrderId, ItemId\}, \{\}$ $SelectItemQuantity, \{OrderId, ItemId, ItemQuantity\}, \{\})^*$ $CloseOrder, \{OrderId, CreditCardNumber\}, \{\}$
$GetConfirmation, \{OrderId, ProductItem\}, \{\}$	$\overline{ConfirmItem}, \{OrderId, ItemId, ItemQuantity\}, \{\}$
$GetConfirmationAll, \{OrderId, ProductItemList\}, \{\}$	$\overline{ConfirmItem}, \{OrderId, ItemId, ItemQuantity\}, \{\}^*$
$\overline{Quit}, \{OrderId\}, \{OrderResult\}$	$CloseOrder, \{\}, \{OrderResult\}$
—	$\overline{PayThirdParty}, \{CreditCardNumber, Amount\}, \{OrderResult\}$

Table C.1: Inferred ontological correspondences

The presented case study highlights that, although implementing complementary functionalities, Blue cannot communicate with Moon due to several behavioral mismatches and a mediator is needed to provide a solution. For instance, within the Blue protocol, there is the possibility to add first all items and only once the order is placed, the confirmations are received. While the Moon protocol, each added item is immediately confirmed. Further, the mediator must satisfy the performance requirement imposed by the NSs on the connected system.

C.4 Enhanced CONNECTor Synthesis Approach

Figure C.5 summarizes the process for the *enhanced* CONNECT*or synthesis*, one of the contributions of this paper. The process combines: (i) the mediator synthesis taking into account the functional concerns (upper side of the figure), and (ii) the performance analysis-based reasoning acting on the intermediary mediator to meet the composed systems' performance requirements (bottom side).

In [59, 60] it is proposed an approach for (i) the automated synthesis of mediators that overcomes interoperability problems between two heterogeneous emerging protocols, given the NSs models, ontology describing the domain-specific knowledge -and a bound on the number of executions of cycles making the traces finite for performance analysis purposes.

The approach consist of three phases or steps: abstraction, matching, and synthesis. The *Abstraction* (❶ in Figure C.5) takes as input the NSs models and the subset of the domain ontology they refer to, and identifies the NSs common language through the ontologies. The common language makes NSs behavior comparable to reason on them. The *Matching* (❷ in Figure C.5) checks the NSs behavioral compatibility, i.e., that the two systems can synchronize at least on one trace reaching one of their respective final state. This step identifies possible mismatches to be reconciled while also taking into account a goal (if specified). Finally, the *Synthesis* (❸ in Figure C.5) produces a (intermediary) mediator that addresses the identified mismatches between the two NSs.

By considering Blue eLTSs (Figure C.3), Moon eLTS (Figure C.4), and the ontological correspondences identifying the common language (Table C.1), the above described phases synthesize the mediator illustrated in Figure C.6.

The performance analysis-based reasoning (ii) enhances our previous approach in order to also take into consideration performance concerns during the synthesis. This is done acting on the intermediary mediator (produced before). For the analysis we use the Æmilia Architectural Description Language (ADL) [15], based on the stochastic process algebra $EMPA_{gr}$ [14]. That provides a formal architectural description of complex software systems allowing the performance analysis of the specified system. We use Æmilia instead of other stochastic formal models (such as Markov Chain) for two reasons: it is easy to generate Æmilia textual description from eLTS and PMM models that are the input to the synthesis approach; and Æmilia ADL is a formalism that maintains the same architectural abstraction of the NSs specification making easier the interpretation of the performance analysis and the reasoning on how to

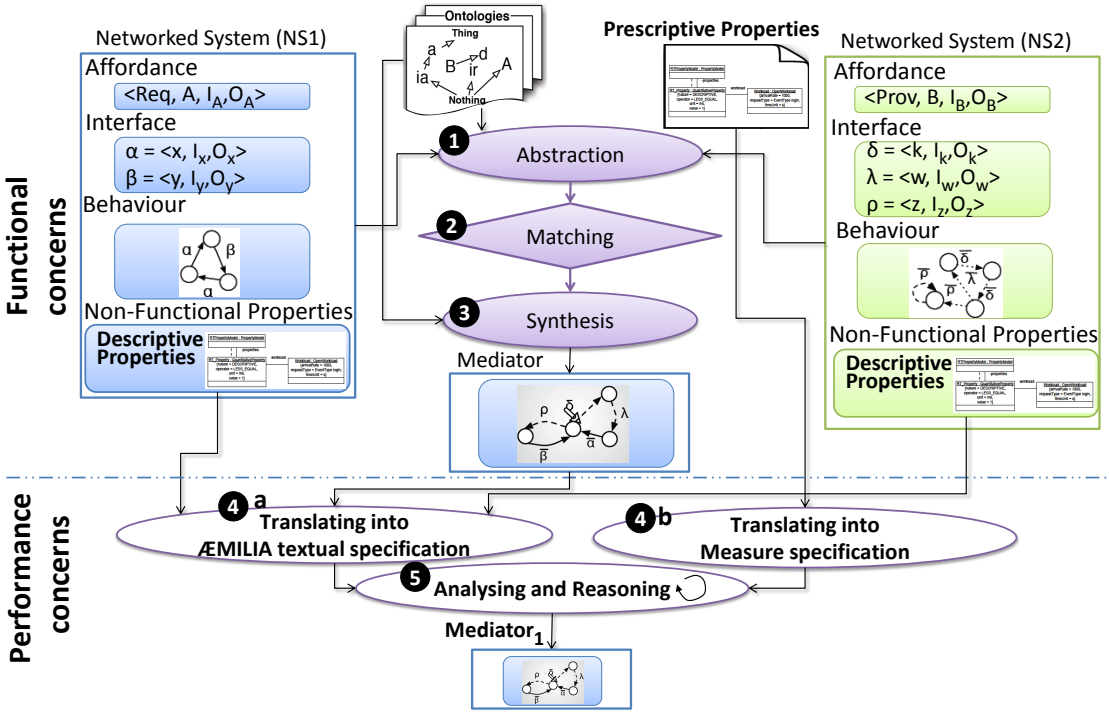

Figure C.5: Overview of the CONNECTor synthesis

refine the mediator to improve the performance of the connected system. The performance analysis-based reasoning is composed by two steps detailed in the following.

Generation of an Æmilia specification. It is composed by two activities: the first one, Figure C.5 ❹**a**, takes as input the eLTS of the NSs and of the synthesized mediator, and the *descriptive properties* to generate the *ÆMILIA textual description*. While the eLTS are used to specify the component behavior in Æmilia ADL, the descriptive properties, expressed as PMM models and specifying the performance characterization of the NSs' actions, are used to set the action rates in the Æmilia specification. To execute performance analysis, the Æmilia analysis tool, named TwoTowers [13], requires a *measure specification* that defines the performance indices of interest. The second activity, Figure C.5 ❹**b**, translates the *prescriptive property*, again expressed as PMM models and defining the performance requirements on the final system, into the required *measure specification*.

Performance analysis-based reasoning. From the ÆMILIA description and the measure specification, the performance analysis is executed. If the composed system satisfies the specified requirements then the mediator is not modified, otherwise it undergoes a reasoning step that tries to obtain a mediator showing better performance (see Figure C.5 ❺). As mentioned before, to improve the composed system performance we can act on the mediator to slice alternative behaviors, limit the number of execution of cycles, and find out the best deployment. At the end of this step two scenarios are possible: i) we obtain a mediator (possibly the initial one) that allows the composed system to meet the performance requirements; ii) all the refined mediators obtained by applying the identified strategies do not allow the composed system to meet the performance requirements. In this case the approach does not produce a mediator and asks to relax the performance requirement in order to provide a suitable system.

Given that Blue-Moon system, all the three strategies can be applied: the deployment strategy is applicable since we assume in CONNECT that the organizations managing the involved NSs agree to deploy the mediator on their side to allow the NSs interoperability; the slicing strategy is feasible since Blue NS presents two alternative interaction behaviors and hence the mediator communication protocol can be sliced accordingly allowing the interactions with Blue via one of the two alternatives; finally for what concerns the upperbound of cycles iterations, it is possible to bound the maximum number of items composing an order.

C.5 Detailing the Enhancement of the CONNECTor Synthesis

This section details the generation of an Æmilia specification (Section C) and the performance analysis-based reasoning step (Section D).

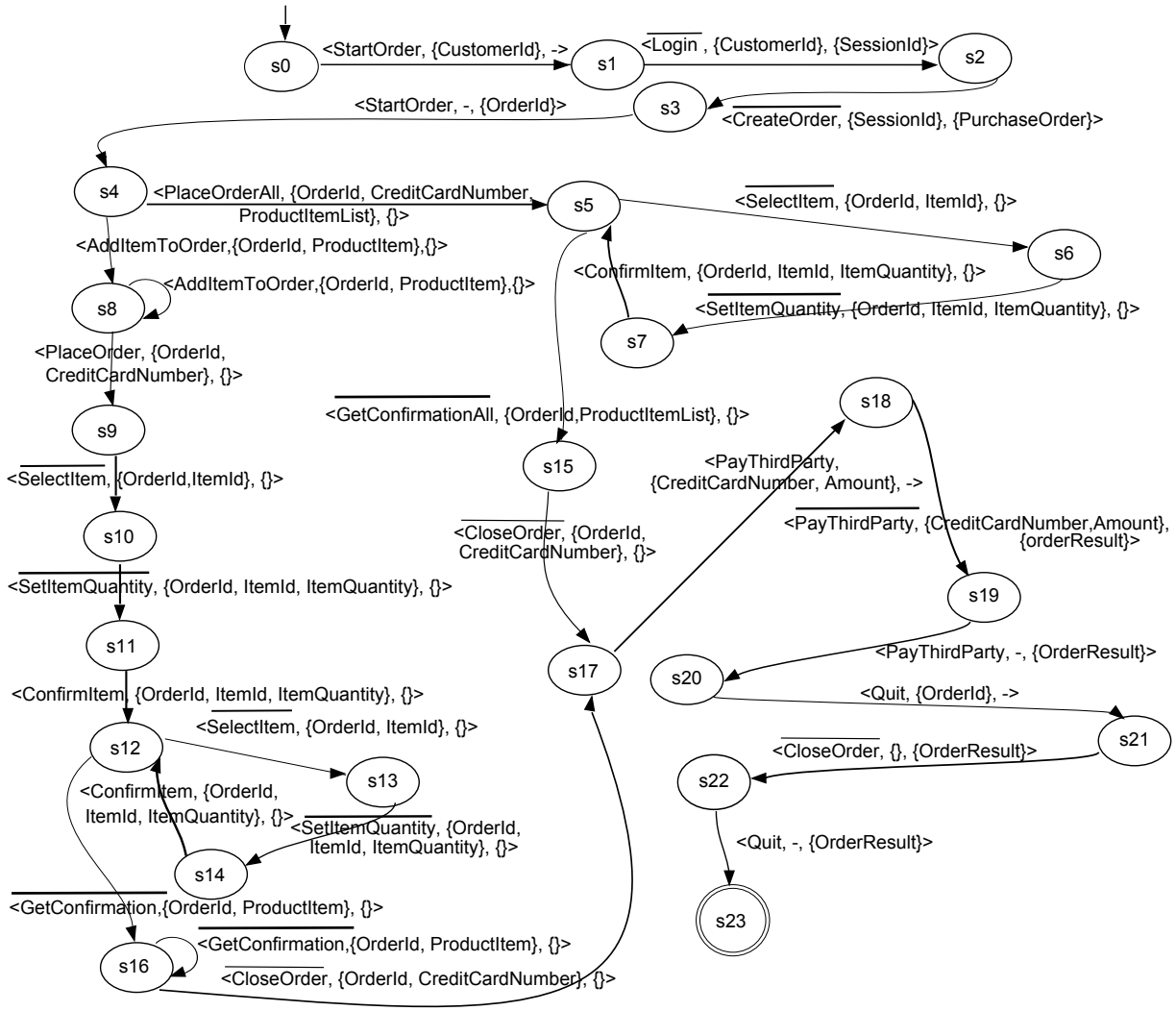

Figure C.6: Mediator Protocol between Blue and Moon

In an Æmilia specification there are three main concepts: (i) *Architectural Type*, that contains the name and the specification of formal parameters of the architectural type; (ii) *Architectural Element Type* (AET), i.e. a component or a connector type that describes the internal behaviors and the interactions of the component; (iii) *Architectural Topology* contains the specification of *Architectural Element Instance* (AEI), i.e. an instance of a given AET type, their *Architectural Interactions* and *Architectural Attachments*, i.e. communication links among instances representing the synchronization actions.

An Æmilia specification can be analyzed by the TwoTowers tool [13] that enables functional verification via model checking or equivalence checking, as well as performance evaluation through the numerical solution of continuous-time Markov chains [107] or discrete-event simulation [114].

C.5.1 ÆMILIA and indices Specification Generation

In this section, we describe how to obtain the specification of Æmilia and of the performance indices of interest, steps ❹**a** and ❹**b** of the process of Figure C.5.

An Æmilia textual description is made by *unparametrized AET*, that we derive from the eLTSs of NSs and mediator, and by *action rate in the AET* from the descriptive properties provided by the NSs for the actions of their interfaces. The performance indices are derived from the prescriptive properties.

Each input action in the (NS/Mediator) eLTS produces in the AET behavior a passive action possibly followed by an exponential action. This auxiliary action is generated when the descriptive response time or service rate property for the input action is provided. For example, we recall that Figure C.2.a shows the response time property (1 ms) of the action *login* provided by the Moon system. From this information we are able to set the auxiliary action rate as the inverse of the response time. The passive action is finally declared as *INPUT_INTERACTION* in the AET specification.

Each output action in the eLTS instead is modeled as non-passive action and it is declared as *OUTPUT_INTERACTION* in the AET specification. The rate of such actions may be either infinite, to model local communications consuming no time, or exponential to model remote communications consuming the time corresponding to the considered rate. Similarly to the eLTS input action, auxiliary exponential action can be inserted in case a descriptive response time or service rate is provided for it.

Finally, the Æmilia Architectural Topology is easy to derive since: *i)* the connected system is composed by one instance for each NS involved and one for the synthesized mediator, *ii)* the attachments are created by connecting the output interactions of the NSs with the homonomous input interactions of the Mediator, and by connecting the output interactions of the Mediator with the homonomous input ones of the NSs.

Listing C.1 reports the derived Æmilia AET for Moon corresponding to the eLTS in Figure C.4. Note that all the actions corresponding to eLTS output actions are here exponential with rate *net*, since in this model we assume that the communications outcoming the Moon system are all remote.

```
ELEM_TYPE Moon(const rate mu_order, const rate mu_item, const rate mu_login_COS, const rate
    mu_prepareCloseOrder, const rate net)

BEHAVIOR
CS0(void;void)=<login,_>.<loginProcess,exp(mu_login_COS)>.<createOrder,_>.
               <createOrderElaboration,exp(mu_order)>.<selectItem,_>.CS3();
CS3(void;void)=<setItemQuantity,_>.<itemProc, exp(mu_item)>.<confirmItem,exp(net)>.choice{
               <selectItem,_>.CS3(),
               <closeOrderCC,_>.<PayThirdParty,exp(net)>.<prepareCloseOrder,exp(mu_prepareCloseOrder)
                    >.<CloseOrder,exp(net)>.CS0()}

INPUT_INTERACTIONS UNI login; createOrder; selectItem; setItemQuantity; closeOrderCC; CloseOrder;
    PayThirdParty_Result
OUTPUT_INTERACTIONS  UNI confirmItem; PayThirdParty; CloseOrder
```

Listing C.1: Æmilia Specification for the Moon Customer System.

For what concerns the indices specification, indication about the performance indices of interest is obtained from the Prescriptive Property defining the performance requirement the connected system must satisfy. We recall that in our case study the requirement on the system throughput is modeled by the property in Figure C.2.b. From this it is possible to map the reward specification for the throughput as required by the TwoTowers and showed in Listing C.2. In this case, the referent action for throughput is the *startOrder* of the Blue instance *B*.

```
MEASURE TR_SYSTEM IS ENABLED (B.startOrder) -> TRANS_REWARD(1);
```

Listing C.2: Throughput specification.

Up to now, the generation algorithm has been designed and not implemented. Its implementation is straightforward since we have deep knowledge about model-to-model transformations involving Æmilia [38]. We plan to do it in the near future.

C.5.2 Analysis and Reasoning

In this section, we report on the analysis and reasoning activities we conducted on the Blue-Moon case study to evaluate the effectiveness of our approach. We run the analysis on a Sony Vaio laptop with a 2 GHz processor, 4 GB of memory, and Window 7 operating system. The TwoTowers tool has been installed on Linux Virtual Machine running Ubuntu distribution. In this setting, all the analysis completed within 1 second demonstrating that it is feasible to use our approach based on Æmilia ADL and TwoTowers tool to enhance mediator synthesis in case of final system size is comparable to the one we here analyze.

We recall that the performance requirement for the Blue-Moon system is: the system must guarantee a throughput that is in average greater than 7 requests/second when it receives a workload of 10 requests per second.

We firstly analyze the assembled Blue-Mediator-Moon system that uses the synthesized Mediator allowing the complete communication with Blue client considering that the three components are deployed on three different devices, hence all the communications between the Mediator and the two NSs are remote and consuming time on the network. In this scenario, the predicted system throughput is 0.604 request/sec that is far to be 7 request/sec as specified in the performance constraint requiring an iteration of reasoning and analysis on the system guided by the strategies we identify in Section C: i) alternative Blue/Mediator behaviors slicing, ii) tuning the upper bound of number of loop iterations, and iii) deployment configuration.

We recall that all the strategies are applicable to the Blue-Moon case studies. Acting on i), since Blue NS has two alternative behaviors, we will consider three versions of the Mediator and hence of the connected system, that is we identify *three scenarios*: **Sys3** - the Blue-Moon system connected via the original synthesized mediator (Figure C.6) that allows to interact with Blue by using its complete communication protocol (Figure C.3); **Sys2** - the Blue-Moon system connected via the sliced mediator permitting the interaction with Blue by using its alternative behavior that places the order with all the items together; **Sys1** - the Blue-Moon system using the sliced mediator allowing the interaction with Blue through its alternative behavior that adds one item at a time and subsequently places the order. Acting on ii) we identify *six scenarios* by imposing six different upper bounds (5, 10, 20, 30, 40 and 50) on the number of items to be added to a purchase order. Acting on iii) we identify *three deployment scenarios*: *remote* where the mediator is deployed remote to both the devices hosting Blue and Moon (three devices), *local to Blue*, where the Mediator is deployed on the same device running Blue removing network latency in the Blue-Mediator communications, and *local to Moon* where the Mediator is local to Moon NS resetting the network latency in their communications.

Combining the above described scenarios, we have 54 final configurations to consider and hence 54 corresponding experiments to conduct. The results of such experiments are all reported in the 3 tables of Figure C.7 where each table refers to one of the mediator deployment scenarios. In each table we have: on the rows, the upper bounds on the number of items to be added in a purchase order, whereas on the columns, which Blue-Mediator configuration we consider in the composed system. A generic cell of the table describes the average composed system throughput given a specific Blue-Mediator configuration (either Sys1, or Sys2, or Sys3), a maximum number of items per order (either max 5, or 10, or 20, or 30, or 40, or 50), and a specific deployment scenario (remote deployment, local to Blue, or local to Moon).

Mediator deployed *remotely*

	Blue3	*Blue2*	*Blue1*
max 50 items	0,604	0,753	0,537
max 40 items	0,732	0,907	0,650
max 30 items	0,929	1,139	0,823
max 20 items	1,271	1,531	1,123
max 10 items	2,013	2,336	1,769
max 5 items	2,650	2,958	2,399

Mediator deployed *locally to Blue*

	Blue3	*Blue2*	*Blue1*
max 50 items	0,846	0,788	0,891
max 40 items	1,029	0,958	1,078
max 30 items	1,312	1,221	1,365
max 20 items	1,811	1,683	1,859
max 10 items	2,922	2,710	2,916
max 5 items	3,958	3,584	3,940

Mediator deployed *locally to Moon*

	Blue3	*Blue2*	*Blue1*
max 50 items	1,997	4,292	1,302
max 40 items	2,377	4,926	1,566
max 30 items	2,934	5,780	1,966
max 20 items	3,832	6,393	2,639
max 10 items	5,526	**8,849**	4,017
max 5 items	6,611	**8,849**	5,276

Figure C.7: Average Throughput Analysis Results

The experiments confirm that the worst deployment is the one that deploys the Mediator remote to both the two NSs. This is intuitive since it introduces overhead (i.e. the network latency) due to remote communication among the NSs and the Mediator. The best deployment is the one that deploys the Mediator local to the Moon NS, since this scenario decreases the communication overhead among the Mediator and Moon that is in average higher than the overhead of the remote communication between Blue and the Mediator. This is the effect of the communication protocols that Blue and Moon expose: Moon always requires the selection of an item at time by using two separate communications, selectItem and selectItemQta; while Blue either communicates the selected items one at time by using one communication, AddItemToOrder, or sends the whole order in one communication using PlaceOrderAll interaction.

The experiments also show that the higher the upper bound of the number of item the worst the system throughput is. This is intuitive since the system execution time for an order processing increases at the number of item composing the purchase order, and hence the system throughput decreases. To have the best throughput in each configuration this upper bound should be set at most to 10 item/order.

Finally, the experiments show that the composed system involving Sys2 performs always better than the others. This is because it implements the behavior that adds all the items at a time to the purchase order, reducing the number of communications between itself and the mediator.

The best configuration that satisfies the performance requirement is the Sys2 system, whose order item upper bound is set either to 5 or 10 and the Mediator is deployed locally to the Moon NS. In this case the system throughput is 8.849 requests/sec, higher than the required limit. In this case, the mediator satisfies both functional and performance requirements for our case study. However, whenever for some reason the Sys2 sliced system is not acceptable, we could propose to relax the performance requirement reducing the required throughput from 7 to 6 request per second and suggest to use the complete synthesized mediator (Sys3), with max 5 items per order, that is still deployed locally to Moon NS.

C.6 Related Work

A big effort has been devoted in the literature to the investigation of the interoperability problem. The theory of mediator, on which we build upon, is closely related to the seminal paper by Yellin and Strom on protocol adaptor synthesis [119]. They propose an adaptor theory to characterize and solve the interoperability problem of augmented interfaces of applications. Yellin and Strom formally define the checks of applications compatibility and the concept of adapters. Furthermore, they provide a theory for the automated generation of adapters based on interface mapping rules, which is related to our common language of protocols found through the domain ontology.

In more recent years an increasing attention has been payed in the Web Service area where many works are related to our synthesis of mediators for some aspect [36], [110], [117], [96]. Among them, papers [62, 63] propose a formal model to describe services and adapters and to automatically generate adapters. The work [52] presents an approach to specify and synthesize adapters based on domain-specific transformation rules and by using existing controller synthesis algorithms implemented in the Marlene tool[3]. The paper [91] on behavioral adaptation proposes a matching approach based on heuristic algorithms to match services for the adapter generation taking into account both the interfaces and the behavioral descriptions. Moreover, the Web services community has been also investigating how to actually support service substitution to enable interoperability with different implementations of a service (e.g., due to evolution or provision by different vendors). Our mediator synthesis work relates, for instance, to [33] by sharing the exploitation of ontology to reason about interface mapping and the synthesis according to such mapping. Their approach can be seen as an instance of ours.

Over the last decade, a lot of research has been directed toward integrating performance analysis into the software development process [39]. All such approaches share the idea of generating the performance models from the software specification/modeling. The performance models are then evaluated to guide the developers in the decisions they have to take during the software life-cycle. Such analysis helps in developing software systems satisfying their performance requirements. This methodology has been applied to several types of system such as Component-based system [70] and software services [3]. To the best of our knowledge none has used this kind of approach to enhance connector/mediator synthesis, hence, our work is original in this sense.

Concerning combined approaches taking into account both functional and non functional issues, we can mention papers [95] and [109]. This latter proposes an approach to automatically derive adaptors in order to assemble correct by construction real-time systems from COTS. The approach takes into account interaction protocols, timing information, and QoS constraints to prevent deadlocks and unbounded buffers. The synthesized adaptor is then a component that mediates the interaction between the components it supervises, in order to harmonize their communication. The purpose of our approach is similar to that in [109] since we both aim at synthesizing a mediator reconciling protocols, but our setting is quite different with respect to theirs. Indeed, our focus is mainly on solving protocols discrepancies to allow protocols synchronization satisfying performance requirements, while they focus more on timing and deadlock issues while composing COTS real-time components.

C.7 Conclusion

The huge number of heterogeneous systems dynamically available within the current networked environment represent a strength in the case there is the possibility to seamlessly exploit systems' capabilities to achieve some purpose.

[3] http://service-technology.org/tools/marlene

This means that there is the need to be able to dynamically manage heterogeneity so to make it an advantage rather than an obstacle.

The CONNECT European project aims to overcome interoperability barriers between heterogeneous protocols by using an approach that dynamically generates the interoperability solution to let them interact. Towards this direction, we proposed an approach to the automated mediator synthesis coping with the functional facet of the problem.

In this paper we illustrated an enhanced CONNECTors synthesis that allows to deal with both functional and non functional aspects. We showed the feasibility of the enhanced synthesis through a running example that we also used to assess the approach effectiveness.

At the moment, the presented approach has been only specified. However, it can be fully automatized if in the NSs: alternative behaviors implementing the same functionality are explicitly specified; the upper bounds of cycles iterations that can be tuned are clearly identified; and available deployment alternatives are explicitly coded, for example by means of a deployment matrix. In this case, it is possible to automatically extract several system alternatives. This would lead to the synthesis of several versions of mediator, by slicing or refining the initially synthesized mediator. From each alternative, the Æmilia description generation and analysis can be performed. On the basis of such results, the better mediator can be easily selected by comparing the analysis results. We plan to implement the whole process as future work.

D Achieving Interoperability through Semantics-based Technologies: The Instant Messaging Case

Amel Bennaceur (Inria), Valérie Issarny (Inria), Romina Spalazzese (UNIVAQ), and Shashank Tyagi (Institute of Technology, Banaras Hindu University, India)

Abstract The success of pervasive computing depends on the ability to compose a multitude of networked applications dynamically in order to achieve user goals. However, applications from different providers are not able to interoperate due to incompatible interaction protocols or disparate data models. Instant messaging is a representative example of the current situation, where various competing applications keep emerging. To enforce interoperability at runtime and in a non-intrusive manner, *mediators* are used to perform the necessary translations and coordination between the heterogeneous applications. Nevertheless, the design of mediators requires considerable knowledge about each application as well as a substantial development effort. In this paper we present an approach based on ontology reasoning and model checking in order to generate correct-by-construction mediators automatically. We demonstrate the feasibility of our approach through a prototype tool and show that it synthesises mediators that achieve efficient interoperation of instant messaging applications.

D.1 Introduction

Pervasive computing promises a future where a multitude of networked applications dynamically discover one another and seamlessly interconnect in order to achieve innovative services. However, this vision is hampered by a plethora of independently-developed applications with *compatible functionalities* but which are unable to interoperate as they realise them using disparate interfaces (data and operations) and protocols. Compatible functionalities means that at a high enough level of abstraction, the functionality provided by one application is semantically equivalent to that required by the other.

The evolution of instant messaging (IM) applications provides a valuable insight into the challenges facing interoperability between today's communicating applications. Indeed, the number of IM users is constantly growing – from around 1.2 billion in 2011 to a predicted 1.6 billion in 2014 [108] – with an increasing emphasis on mobility – 11% of desktop computers and 18% of smartphones have instant messaging applications installed [92]– and the scope of IM providers is expanding to include social networking such as Facebook that embeds native IM services onto their Web site. Consequently, different versions and competing standards continue to emerge. Although this situation may be frustrating from a user perspective, it seems unlikely to change. Therefore, many solutions that aggregate the disparate systems, without rewriting or modifying them, have been proposed [61]. These solutions use intermediary middleware entities, called *mediators* [115] – also called mediating adapters [119], or converters [31] – which perform the necessary coordination and translations to allow applications to interoperate despite the heterogeneity of their data models and interaction protocols.

Nevertheless, creating mediators requires a substantial development effort and thorough knowledge of the application-domain. Moreover, the increasing complexity of today's software systems, sometimes referred to as Systems of Systems [79], makes it almost impossible to manually develop 'correct' mediators, i.e., mediators guaranteeing deadlock-free interactions and the absence of unspecified receptions [119]. Starlink [26] assists developers in this task by providing a framework that performs the necessary mediation based on a domain-specific description of the translation logic. Although this approach facilitates the development of mediators, developers are still required to understand both systems to be bridged and to specify the translations.

Furthermore, in pervasive environments where there is no *a priori* knowledge about the concrete applications to be connected, it is essential to guarantee that the applications associate the same *meaning* to the data they exchange, i.e., semantic interoperability [56]. *Ontologies* support semantic interoperability by providing a machine-interpretable means to automatically reason about the meaning of data based on the shared understanding of the application domain [6]. Ontologies have been proposed for Instant Messaging although not for the sake of protocol interoperability but rather for semantic archiving and enhanced content management [47]. In a broader context, ontologies have also been widely used for the modelling of Semantic Web Services, and to achieve efficient service discovery and composition [87]. Semantic Markup for Web Services[1] (OWL-S) uses ontologies to model both the functionality and the behaviour of Web services. Besides semantic modelling, Web Service modelling Ontology (WSMO) supports runtime mediation based on pre-defined mediation patterns but without ensuring that such mediation does not lead to a deadlock [36]. Although ontologies have long been advocated as a key enabler in the context of service mediation, no principled approach has been proposed yet to the automated synthesis of mediators by systematically exploiting ontologies [19].

[1] http://www.w3.org/Submission/OWL-S/

This paper focuses on distributed applications that exhibit compatible functionalities but are unable to interact successfully due to mismatching interfaces or protocols. We present an approach to synthesise mediators automatically to ensure the interoperation of heterogeneous applications based on the semantic compatibility of their data and operations. Specifically, we rely on a domain-specific ontology (e.g., an IM ontology) to infer one-to-one mappings between the operations of the applications' interfaces and exploit these mappings to generate a correct-by-construction mediator. Our contribution is threefold:

- *Formal modelling of interaction protocols.* We introduce an ontology-based process algebra, which we call Ontology-based Finite State Processes (OFSP), to describe the observable behaviour of applications. The rationale behind a formal specification is to make precise and rigorous the description and the automated analysis of the observable behaviour of applications.
- *Automated generation of mediators for distributed systems.* We reason about the semantics of data and operations of each application and use a domain ontology to establish, if they exist, one-to-one mappings between the operations of their interfaces. Then, we verify that these mappings guarantee the correct interaction of the two applications and we generate the corresponding mediator.
- *Framework for automated mediation.* We provide a framework that refines the synthesised mediator and deploys it in order to automatically translate and coordinate the messages of mediated applications.

Section D examines in more detail the challenges to interoperability using the IM case. Section D introduces the ontology-based model used to specify the interaction protocols of application. Section D presents our approach to the automated synthesis of mediators that overcome data and protocol mismatches of functionally compatible applications and illustrates it using heterogeneous instant messaging applications. Section D describes the tool implementation while Section D reports the experiments we conducted with the instant messaging applications and evaluate the approach. The results show that our solution significantly reduces the programming effort and ensures the correctness of the mediation while preserving efficient execution time. Section D examines related work. Finally, Section D concludes the paper and discusses future work.

D.2 The Instant Messaging Case

Instant messaging (IM) is a popular application for many Internet users and is now even embedded in many social networking systems such as Facebook. Moreover, since IM allows users to communicate in real-time and increases their collaboration, it is suitable for short-lived events and conferences such as Instant Communities for online interaction at the European Future Technologies Conference and Exhibition[2] (FET'11) that took place in May 2011.

Popular and widespread IM applications include Windows Live Messenger[3](commonly called MSN messenger), Yahoo! Messenger[4], and Google Talk[5] which is based on the Extensible Messaging and Presence Protocol[6] (XMPP) standard protocol. These IM applications offer similar functionalities such as managing a list of contacts or exchanging textual messages. However, a user of Yahoo! Messenger is unable to exchange instant messages with a user of Google Talk. Indeed, there is no common standard for IM. Thus, users have to maintain multiple accounts in order to interact with each other (see Figure D.1). This situation, though cumbersome from a user perspective, unfortunately reflects the way IM – like many other existing applications – has developed.

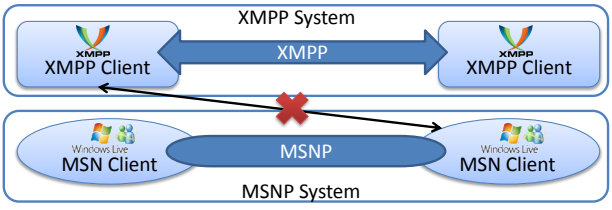

Figure D.1: Interoperability issue between heterogeneous IM systems

A solution that guarantees interoperability between heterogeneous IM applications has to cope with the following heterogeneity dimensions:

[2] http://www.fet11.eu/
[3] http://explore.live.com/windows-live-messenger/
[4] http://messenger.yahoo.com/
[5] http://www.google.com/talk/
[6] http://www.xmpp.org/

- *Data heterogeneity*. MSN Messenger protocol (MSNP), the protocol used by Windows Live Messenger, uses text-based messages whose structure includes several constants with predefined values. On the other hand, the Yahoo! Messenger Protocol (YMSG) defines binary messages that include a header and key-value pairs. As for XMPP messages, they are defined according to a given XML Schema.
- *Protocol heterogeneity*. Even though IM applications are simple and quite similar, each one communicates with its own proprietary application server used to authenticate and to relay the messages between instant messaging clients. Consequently, each application has its own interaction protocol.

Achieving interoperability between independently developed systems has been one of the fundamental goals of middleware research. Prior efforts have largely concentrated on solutions where conformance to the same standard is required e.g., XMPP. However, compliance to a unique standard is not always feasible given the competitive pressures in the marketplace.

Middleware-based approaches define a common abstraction interface (e.g., Adium[7]) or an intermediary protocol (e.g., J-EAI[8] and CrossTalk [89]) promote interoperability in a transparent manner. However, relying on a fixed intermediary interface or protocol might become restrictive over time as new functionalities and features emerge. By synthesising mediators automatically and rigorously we relieve developers from the burden of implementing or specifying such mediators and further ensures their correctness.

Semantics-based solutions (e.g., SAM [47] and Nabu[9]) use ontologies to enhance the functionalities of IM applications by reasoning about the content of messages and overcoming mismatches at the data level but assume the use of the same underlying communication protocol. Hence, even though an enormous amount of work is being carried out on the development of concrete interoperability solutions that rely on ontologies to overcome application heterogeneity, none propose an approach to generate mediators able to overcome both data and protocol heterogeneity. In the next section, we introduce our ontology-based approach to interoperability that automatically synthesises mediators to transparently solve both data and protocol mismatches between functionally compatible applications at runtime.

D.3 Ontology-based Modelling of Interaction Protocols

Automated mediation of heterogeneous applications requires the adequate modelling of their data and interaction protocols. In this section, we introduce OFSP (Ontology-based Finite State Processes), a semantically-annotated process algebra to model application behaviour.

D.3.1 Ontologies in a Nutshell

An ontology is *a shared, descriptive, structural model, representing reality by a set of concepts, their interrelations, and constraints under the open-world assumption* [6]. The Web Ontology Language[10] (OWL) is a W3C standard language to formally model ontologies in the Semantic Web. Concepts are defined as OWL classes. Relations between classes are called OWL properties. Ontology reasoners are used to support automatic inference on concepts in order to reveal new relations that may not have been recognised by the ontology designers. OWL is based on description logics (DL), which is a knowledge representation formalism with well-understood formal properties [6]. To verify the interaction of networked applications, we are in particular interested in specialisation/generalisation relations between their concepts. In this sense, DL resemble in many ways type systems with concept *subsumption* corresponding to type subsumption. Nevertheless, DL are by design and tradition well-suited for domain-specific services and further facilitate the definition and reasoning about composite concepts, e.g., concepts constructed as disjunction or conjunction of other concepts. Subsumption is the basic reasoning mechanism and can be used to implement other inferences, such as satisfiability and equivalence, using pre-defined reductions [6]:

Definition 4 ($\sqsubseteq$: Subsumption) *A concept C is subsumed by a concept D in a given ontology $\mathcal{O}$, written $C \sqsubseteq D$, if in every world consistent with the axioms of the ontology $\mathcal{O}$ the set denoted by C is a subset of the set denoted by D.*

The subsumption relation is both transitive and reflexive and defines a hierarchy of concepts. This hierarchy always contains a built-in top concept *owl:Thing* and bottom concept *owl:Nothing*.

Figure D.2 depicts the instant messaging ontology. An InstantMessage class has at least one sender hasSender{some}, one recipient hasRecipient{some}, and one message hasMessage. hasSender{some} and hasRecipient{some} are object properties that relate an instant message to a sender or a recipient while hasMessage is a data property associated with the InstantMessage class. The Sender and Recipient classes are subsumed by

[7] http://adium.im/
[8] http://www.process-one.net
[9] http://nabu.opendfki.de/
[10] http://www.w3.org/TR/owl2-overview/

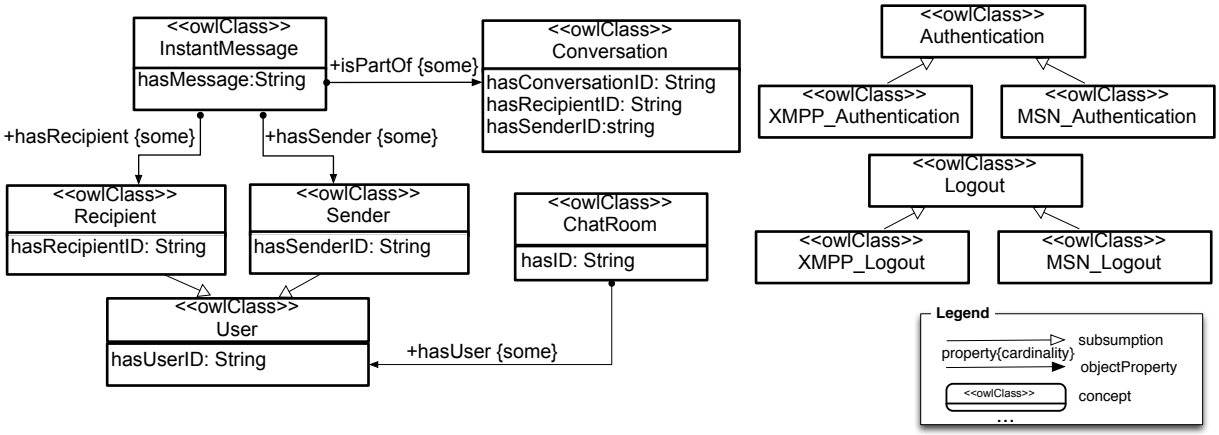

Figure D.2: The instant messaging ontology

the User class. Indeed, any instance of the two former classes is also an instance of the latter. A Conversation is performed between a sender (who initialises it) and a recipient, and the conversation has its own identifier. An instant message isPartOf a conversation. A ChatRoom represents a venue where multiple users can join and exchange messages.

D.3.2 Modelling Protocols using Ontology-based FSP

The interaction protocol of an application describes how the operations of its *interface* are coordinated in order to achieve a specified functionality. We build upon state-of-the-art approaches to formalise interaction protocols using process algebra, in particular Finite State Processes (FSP) [78]. FSP has proven to be a convenient formalism for specifying concurrent systems. Although another process algebra would have worked equally well, we choose FSP for convenience and to exploit the Labelled Transition System Analyser (LTSA) in order to automate reasoning and analysis of interaction protocols specified as finite processes.

Each process P is associated with an interface αP that defines the set of observable actions that the application requires from/provides to its running environment. We structure these actions and annotate them using a domain ontology $\mathcal{O}$ so as to specify their semantics, resulting in Ontology-based FSP (OFSP). An *input action* $a =<op, I, O>$ specifies a required operation $op \in \mathcal{O}$ for which the application produces a set of input data $I = \{in \in \mathcal{O}\}$ and consumes a set of output data $O = \{out \in \mathcal{O}\}$. The dual *output action*[11] $\overline{b} =<\overline{op}, I, O>$ refers to a provided operation op for which the application uses the inputs I and produces the corresponding outputs O. Note that all actions are annotated using the same domain ontology $\mathcal{O}$ describing the application-specific concepts and relations. The rationale behind this notation is to enable behavioural analysis based on the semantics of process actions. Indeed, only if both sides of communication assign the same semantics to their actions, can they interact correctly. In addition, τ is used to denote an internal action that cannot be observed by the environment. There are two types of processes: *primitive processes* and *composite processes*. Primitive processes are constructed through action prefix ($\rightarrow$), choice ($|$), and sequential composition ($;$). Composite processes are constructed using parallel composition ($\|$).

The semantics of OFSP builds upon the semantics of FSP, which is given in terms of Labelled Transition Systems (LTS) [66]. The LTS interpreting an OFSP process P is a directed graph whose nodes represent the process states and each edge is labelled with an action $a \in \alpha P$ representing the behaviour of P after it engages in an action a. $P \xrightarrow{a} P'$ denotes that P transits with action a into P'. $P \stackrel{s}{\Rightarrow} P'$ is a shorthand for $P \xrightarrow{a_1} P_1 ... \xrightarrow{a_n} P', s = \langle a_1, ..., a_n \rangle, a_i \in \alpha P \cup \tau$. There exists a start node from which the process begins its execution. The END state indicates a successful termination. $traces(P)$ denotes the set of all successfully-terminating traces of P. When composed in parallel, processes synchronise on dual actions while actions that are in the alphabet of only one of the two processes can occur independently of the other process.

The concepts and properties defined in the IM ontology are used to specify MSNP and XMPP clients using OFSP, as illustrated in Figures D.3 and D.4 respectively, focusing on message exchange. Each IM application performs authentication and logout with the associated server. Before exchanging messages, the MSNP application has to configure a *chat room* where the MSN conversation can take place between the user that initiates this conversation (sender) and the user who accepts to participate in this conversation (recipient). In XMPP each message simply contains both the sender and the recipient identifiers.

[11] Note the use of an overline as a convenient shorthand notation to denote output actions

$$
\begin{aligned}
\text{MSNClient} &= (< MSN\_Authentication\_Request, \{UserID\}, \{Challenge\} > \\
&\quad \rightarrow < MSN\_Authentication\_Response, \{Response\}, \{Authentication\_ok\} > \\
&\quad \rightarrow \text{ExchangeMsgs}). \\
\text{ExchangeMsgs} &= (< CreateChatRoom, \{UserID\}, \{ConversationID\} > \\
&\quad \rightarrow < JoinChatRoom, \{UserID\}, \{Acceptance\} > \rightarrow \text{P1} \\
&\quad | < \overline{JoinChatRoom}, \{UserID\}, \{Acceptance\} > \\
&\quad \rightarrow < \{ChatRoomInfo, \emptyset, \{ConversationID\} > \rightarrow \text{P1}), \\
\text{P1} &= (< InstantMessage, \{UserID, ConversationID, Message\}, \emptyset > \rightarrow \text{P1} \\
&\quad | < \overline{InstantMessage}, \{UserID, ConversationID, Message\}, \emptyset > \rightarrow \text{P1} \\
&\quad | < MSN\_Logout, \{UserID\}, \emptyset > \rightarrow \text{END}).
\end{aligned}
$$

Figure D.3: OFSP specification of MSNP

$$
\begin{aligned}
\text{XMPPClient} &= (< XMPP\_Authentication\_Request, \{UserID\}, \{Challenge\ \} > \\
&\quad \rightarrow < XMPP\_Authentication\_Response, \{Response\}, \{Authentication\_ok\} > \\
&\quad \rightarrow \text{ExchangeMsgs}). \\
\text{ExchangeMsgs} &= (< InstantMessage, \{SenderID, RecepientID, Message\}, \emptyset > \rightarrow \text{ExchangeMsgs} \\
&\quad | < \overline{InstantMessage}, \{SenderID, RecipientID, Message\}, emptyset > \\
&\quad \rightarrow \text{ExchangeMsgs} \\
&\quad | < XMPP\_Logout, \{UserID\}, emptyset > \rightarrow \text{END}).
\end{aligned}
$$

Figure D.4: OFSP specification of XMPP

D.4 Ontology-based Approach to Mediator Synthesis

In this section we consider two functionally-compatible applications, described through OFSP processes P_1 and P_2, that are unable to interoperate due to differences in their interfaces or protocols. Functional compatibility means that their required/provided high-level functionalities are semantically equivalent [61]. Our aim is to enforce their interoperation by synthesising a mediator that addresses these differences and guarantees their *behavioural matching*. The notion of behavioural matching is formally captured through *refinement* [57]. A process Q *refines* a process P if every trace of Q is also a trace of P, i.e., $traces(Q) \subseteq traces(P)$. However, this notion of refinement analyses the dynamic behaviour of processes assuming close-world settings, i.e., the use of the same interface to define the actions of both processes. What is needed is a notion of compatibility that takes into account the semantics of actions while relying on a mediator process M to compensate for the syntactic differences between actions and guarantees that the processes communicate properly.

To this end, we first reason about the semantics of actions so as to infer the correspondences between the actions of the processes' interfaces and generate the mapping processes that perform the necessary translations between *semantically compatible actions*. Various mapping relations may be defined. They primarily differ according to their complexity and inversely proportional flexibility. In this paper we focus on one-to-one mappings, i.e., direct correspondences between actions. During the synthesis step, we explore the various possible mappings in order to produce a correct-by-construction mediator, i.e., a mediator M that guarantees that the composite process $P_1 \| M \| P_2$ reaches an END state, or determines that no such mediator exists.

In this section we introduce the semantic compatibility of actions, and use it to define behavioural matching. Then, we present the automated synthesis algorithm.

D.4.1 Semantic Compatibility of Actions

A *sine qua non* condition for two processes P_1 and P_2 to interact is to agree on the data they exchange. However, independently-developed applications often define different interfaces. The mediator can compensate for the differences between interfaces by mapping their actions if and only if they have the same semantics. We first define the notion of *action subsumption* and then, use it to define the *semantic compatibility* of actions.

Definition 5 ($\sqsubseteq_{\mathcal{O}}$: Action Subsumption) *An action* $a_1 = < op_1, I_1, O_1 >$ *is subsumed by an action* $\overline{a_2} = < \overline{op_2}, I_2, O_2 >$ *according to a given ontology* $\mathcal{O}$*, noted* $a_1 \sqsubseteq_{\mathcal{O}} a_2$*, iff: (i)* $op_2 \sqsubseteq op_1$*, (ii)* $\forall i_2 \in I_2, \exists i_1 \in I_1$ *such that* $i_1 \sqsubseteq i_2$*, and (iii)* $\forall o_1 \in O_1, \exists o_2 \in O_2$ *such that* $o_2 \sqsubseteq o_1$*.*

The idea behind this definition is that an input action can be mapped to an output one if the required operation is less demanding; it provides richer input data and needs less output data. This leads us to the following definition of semantic compatibility of actions:

Definition 6 ($\approx_{\mathcal{O}}$: Semantic Compatibility of Actions) *An action a_1 is semantically compatible with an action a_2, denoted $a_1 \approx_{\mathcal{O}} a_2$, iff a_1 is subsumed by a_2 (i.e., a_1 is required and a_2 provided) or a_2 is subsumed by a_1 (a_2 is required and a_1 provided) .*

The semantic compatibility between two actions allows us to generate an action mapping process as follows:

$$\mathsf{M}_{\mathcal{O}}(a_1, a_2) = \begin{cases} a_1 \overset{\mathcal{O}}{\longmapsto} \overline{a_2} & \text{if} \quad a_1 \text{ is subsumed by } \overline{a_2} \\ a_2 \overset{\mathcal{O}}{\longmapsto} \overline{a_1} & \text{if} \quad a_2 \text{ is subsumed by } \overline{a_1} \end{cases}$$

The process that maps action a_1 to action $\overline{a_2}$, written $a_1 \overset{\mathcal{O}}{\longmapsto} \overline{a_2}$ captures each input data from the input action, assigns it to the appropriate input of the output action ($i_2 \leftarrow i_1$), then takes each output data of the output action and assigns it to the expected output of the input action ($o_1 \leftarrow o_2$). This assignment is safe since an instance of i_1 (resp. o_2) is necessarily an instance i_2 of (resp. o_1).

Let us consider $a_1 = <InstantMessage, \{UserID, ConversationID, Message\}, \emptyset>$ associated to the MSN client and $\overline{a_2} = <\overline{InstantMessage}, \{SenderID, RecipientID, Message\}, \emptyset>$ associated to the XMPP client. The IM ontology indicates that (i) Sender is subsumed by User, and (ii) $ConversationID$ identifies a unique Conversation, which includes a $RecipientID$ attribute. Consequently, a_1 is subsumed by $\overline{a_2}$.

D.4.2 Behavioural Matching through Ontology-based Model Checking

We aim at assessing behavioural matching of two processes P_1 and P_2 given the semantic compatibility of their actions according to an ontology $\mathcal{O}$. To this end, we first filter out communications with third party processes [105]. The communicating trace set of P_1 with P_2, noted $traces(P_1)\uparrow_{\mathcal{O}} P_2$ is the set of all successfully-terminating traces of P_1 restricted to the observable actions that have semantically compatible actions in αP_2.

Definition 7 ($\uparrow_{\mathcal{O}}$: Communicating Trace Set) $traces(P_1)\uparrow_{\mathcal{O}} P_2 \overset{\text{def}}{=}$
$\{s = \langle a_1, a_2, ..., a_n \rangle, a_i \in \alpha P_1 \mid P_1 \overset{s}{\Rightarrow} \text{END}$ *such that* $\forall a_i, \exists b_i \in \alpha P_2 | a_i \approx_{\mathcal{O}} b_i\}$

As an illustration, both the MSNP and XMPP IM clients perform their authentication and logout with their respective servers. Additionally, MSNP also performs the actions related to the configuration of the chat room with its servers. Consequently their communicating traces sets are restricted to instant message exchange.

Then, two traces $s_1 = \langle a_1 a_2 ... a_n \rangle$ and $s_2 = \langle b_1 b_2 ... b_n \rangle$ *semantically match*, written $s_1 \equiv_{\mathcal{O}} s_2$, iff their actions semantically match in sequence.

Definition 8 ($\equiv_{\mathcal{O}}$: Semantically Matching Traces)
$s_1 \equiv_{\mathcal{O}} s_2 \overset{\text{def}}{=} a_i \approx_{\mathcal{O}} b_i \;\; 1 \leq i \leq n$

The associated mapping is then as follows:

$$\mathsf{Map}_{\mathcal{O}}(s_1, s_2) = \mathsf{M}_{\mathcal{O}}(a_1, b_1); ...; \mathsf{M}_{\mathcal{O}}(a_n, b_n)$$

Based on the semantic matching of traces, a process P_2 *ontologically refines* a process P_1 ($P_1 \models_{\mathcal{O}} P_2$) iff each trace of P_2 semantically matches a trace of P_1:

Definition 9 ($\models_{\mathcal{O}}$: Ontological Refinement)

$$P_1 \models_{\mathcal{O}} P_2 \overset{\text{def}}{=} \forall s_2 \in traces(P_2)\uparrow_{\mathcal{O}} P_1, \exists s_1 \in traces(P_1)\uparrow_{\mathcal{O}} P_2 : s_2 \equiv_{\mathcal{O}} s_1$$

By checking ontological refinement between P_1 and P_2, we are able to determine the following *behavioural matching* relations:

- *Exact matching*–$(P_1 \models_{\mathcal{O}} P_2) \wedge (P_2 \models_{\mathcal{O}} P_1)$: assesses compatibility for symmetric interactions such as peer-to-peer communication where both processes provide and require the similar functionality.
- *Plugin matching*–$(P_1 \models_{\mathcal{O}} P_2) \wedge (P_2 \not\models_{\mathcal{O}} P_1)$: evaluates compatibility for asymmetric interactions such as client-server communication where P_1 is providing a functionality required by P_2.
- *No matching*–$(P_1 \not\models_{\mathcal{O}} P_2) \wedge (P_2 \not\models_{\mathcal{O}} P_1)$: identifies behavioural mismatch.

Behavioural matching is automated through *ontology-based model checking*. Model checking is an attractive and appealing approach to ensure system correctness that proved to be a very sound technique to automatically verify concurrent systems. The gist of model checking approaches is the exhaustive state exploration. This exploration is performed by model checkers using efficient algorithms and techniques that make it possible to verify systems of up to 10^{1300} states in few seconds [37]. However, even if these techniques effectively handle very large systems, the actions of the models they consider are usually simple strings and the verification matches actions based on their syntactic equality. We build upon these model checking techniques but further match actions based on their semantic compatibility. The semantic compatibility of actions is defined based on the domain knowledge encoded within a given ontology.

Referring to the IM case, all the traces of MSNP and XMPP processes semantically match. Subsequently, these two processes are in exact matching relation, and a mediator can be synthesised to perform action translations and enable their correct interaction.

D.4.3 Automated Mediator Synthesis

In the case where P_1 and P_2 match, that is exact matching in the case of peer-to-peer communication or plugin matching in the case of client/server communication, we synthesise the mediator that makes them properly interact. The algorithm incrementally builds a mediator M by forcing the two protocols to progress synchronously so that if one requires an action a, the other must provide a semantically compatible action $\bar{b}$. The mediator compensates for the syntactic differences between their actions by performing the necessary transformations, which is formalised as follows:

$$\text{Mediator}_{\mathcal{O}}(P_1, P_2) = \|\text{Map}(s_1, s_2) \text{ such that } s_2 \in traces(P_2)\uparrow_{\mathcal{O}} P_1, s_1 \in traces(P_1)\uparrow_{\mathcal{O}} P_2 : s_2 \equiv_{\mathcal{O}} s_1$$

In the IM case, we are able to produce the mediator for the MSNP and XMPP processes as illustrated in Figure D.5. The mediator intercepts an instant message sent by an MSNP user and forwards it to the appropriate XMPP user. Similarly, each instant message sent by an XMPP user, is forwarded by the mediator to the corresponding MSNP user.

$$\begin{array}{ll} \text{Map}_1 & = (<InstantMessage, \{SenderID, RecepientID, Message\}, \emptyset > \\ & \quad \rightarrow <\overline{InstantMessage}, \{UserID, ConversationID, Message\}, \emptyset > \rightarrow \text{END}). \\ \text{Map}_2 & = (< InstantMessage, \{UserID, ConversationID, Message\}, \emptyset > \\ & \quad \rightarrow <\overline{InstantMessage}, \{SenderID, RecepientID, Message\}, \emptyset > \rightarrow \text{END}). \\ \|\text{Mediator} & = (\text{Map}_1 \| \text{Map}_2). \end{array}$$

Figure D.5: OFSP specification of the Mediator between MSNP and XMPP

D.5 Implementation

In order to validate our approach, we have combined the LTSA[12] model checker with an OWL-based reasoner to achieve ontological refinement leading to the OLTSA tool (Figure D.6-❶). LTSA is a free Java-based verification tool that automatically composes, analyses, graphically animates FSP processes and checks safety and liveness properties against them.

In the case where the processes match, a concrete mediator that implements the actual message translation is deployed atop of the Starlink framework [26], see Figure D.6-❸. Starlink interprets the specification of mediators given in a domain-specific language called Message Translation Logic (MTL). An MTL specification describes a set of *assignment*s between message *field*s. The messages correspond to action names and the fields to the name of input/output data. Note that the OFSP description focuses on the ontological annotations and not the the actual name. Therefore, we refine the OFSP specification of the mediator so as to generate the associated MTL before deploying the mediator atop of Starlink, see Figure D.6-❷.

Let us consider the mapping Map_1 (see Figure D.5), which transforms an XMPP input action to the associated MSNP action. Figure D.7 shows a small fragment of the associated translation logic described in MTL and which corresponds to the assignment of the $UserID$ field of the XMPP message ($ReceivedInstantMessage$) to the $SenderID$ field of of the MSNP message (SDG) with the mediator transiting from state $XS1$ to state $MR1$.

The tool, the IM ontology, and a video demonstration are available at http://www-roc.inria.fr/arles/download/imInteroperability/.

[12] http://www.doc.ic.ac.uk/ltsa/

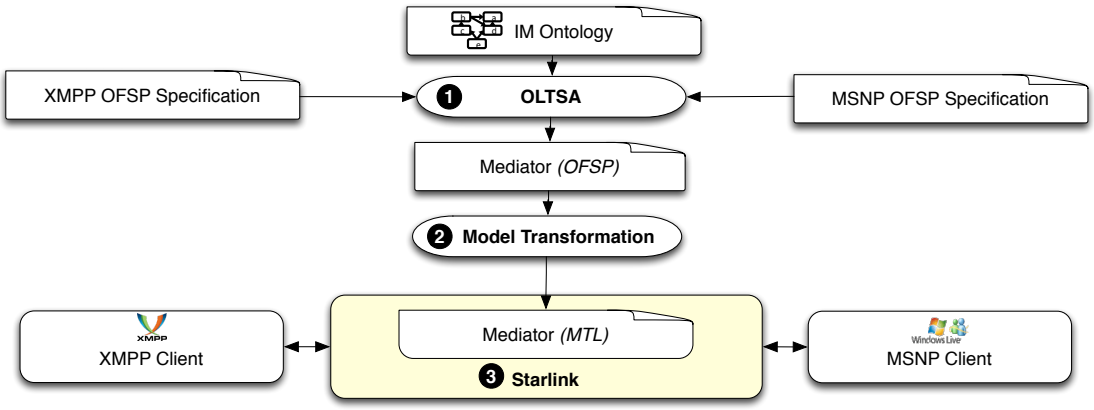

Figure D.6: Mediation Architecture

```
<translationlogic>                                                                    1
 <assignment>                                                                         2
  <field>                                                                             3
   <statelabel>MR1</statelabel><message>SDG</message>                                 4
   <xpath>/field/primitiveField[label='UserID']/value</xpath>                         5
  </field>                                                                            6
  <field>                                                                             7
   <statelabel>XS1</statelabel><message>ReceivedInstantMessage</message>              8
   <xpath>/field/primitiveField[label='SenderID']/value</xpath>                       9
  </field></assignment> ....                                                          10
</translationlogic>                                                                   11
```

Figure D.7: Translation logic to map MSNP and XMPP instant messages

D.6 Assessment

In this section we first report a set of experiments we conducted to evaluate the effectiveness of our approach when applying it to the instant messaging case. Then, we discuss some of its quality properties.

D.6.1 Experimental Results

We have evaluated the time for translating one protocol to the other by the synthesised mediator and the effort required by the developer to enable mediation. We have hand-coded a mediator that makes MSNP, YMSG, XMPP interoperable in order to gauge the complexity of the mediation. We considered the Windows Live Messenger for MSNP, Yahoo! Messenger for YMSG, and Pidgin[13] for XMPP. We run the OLTSA tool and the Starlink framework on a Mac computer with a 2,7 GHz processor and 8 GB of memory.

In the first experiment, we measured the time taken to translate from one protocol to another. We repeated the experiments 50 times and reported the mean time for each case in Table D.1. The hand-coded mediator is approximately 3 times faster than the synthesised one. This is mainly due to the fact that the models are first interpreted then executed by Starlink at runtime whereas the hand-coded mediator is already compiled and hence more efficient.

In the second experiment, we measured the time for synthesising the mediator (see Table D.2). One can note that action mapping is the most time consuming step as it necessitates ontology reasoning in order to infer semantically matching actions while the behavioural matching is performed is less than 1 ms. Nevertheless, this step needs to be performed once only and is definitely faster than hand-coding the mediator or even specifying it. Moreover, for each new version of one of the protocols, the hand-coded mediator has to be re-implemented and re-compiled, Starlink requires the specification of the translation logic to be re-specified whereas the automated synthesis requires only the specification of the protocol to be re-loaded.

	Hand-coded	Mediator atop Starlink
YMSG ↔ MSNP	22	69
MSNP ↔ XMPP	52	131
YMSG ↔ XMPP	44	126

Table D.1: Translation time (ms)

	Act. mapping	Beh. match.
YMSG ↔ MSNP	306	<1
MSNP ↔ XMPP	252	<1
YMSG ↔ XMPP	244	<1

Table D.2: Time for Synthesis (ms)

The third experiment measures the effort demanded from the developer to produce mediators between different IM applications. We calculate the number of Java code lines of the hand-coded mediator, the number of lines of DSL specification that need to be specified for Starlink and those needed to specify the individual applications for the automated synthesis.

	Hand-Coded	Starlink	Automated
YMSG ↔ MSNP	1172	258	96
MSNP ↔ XMPP	750	198	84
YMSG ↔ XMPP	945	168	76

Table D.3: Development effort

The results are given in Table D.3. One can notice that although Starlink reduces considerably (around 4 times) the lines of code that need to be written, the automated approach requires the OFSP specifications only and decreases this number drastically (around 10 times). This is mainly due to (i) the use of OFSP to model the interaction protocols, which introduces an ontology-based domain-specific language grounded in process algebra and especially targeted for concurrent systems. For example, the MSNP behaviour is described in Starlink using 30 XML lines and only 6 lines with our approach (ii) Further, the translation code need not be specified. More importantly, unlike the hand-coded or the Starlink versions where the developer is required to know both protocols and define the translation manually, the protocols are specified separately in the automated version. Thus, each IM provider can independently specify its own protocol. Finally, we are investigating within the CONNECT[14] project learning-based techniques to infer such a specification automatically [19].

[13] http://www.pidgin.im/

[14] http://connect-forever.eu/

To sum up, our automated approach to interoperability significantly reduces the programming effort and ensures the correctness of the translation while requiring a negligible time for synthesising the mediator and guaranteeing good performances at translation time.

D.6.2 Qualitative Assessment

In addition to the above-mentioned performances, our approach satisfies the following properties:

- *Correctness by construction.* The correctness of the mediation, i.e., the absence of deadlock and unspecified receptions [119], is guaranteed by construction. Indeed, if there is an exact match between P_1 and P_2 then the parallel composition $P_1 \| M \| P_2$ is deadlock free. Exact matching means that each trace of P_1 (P_2) has a corresponding semantically-matching trace in P_2 (P_1), which amounts to setting P_1 (P_2) as a safety property that needs to be verified by P_2 (P_1). This verification is performed by exhaustively exploring the state space. Note though that efficient model checkers use optimisation techniques to reduce the space if possible. The reduction techniques are even more efficient in the case of process algebra.
- *Formal yet tractable DSL specification.* OFSP introduces an ontology-based domain-specific language grounded in process algebra. Process algebra constitute a very expressive behavioural specification language for complex concurrent systems while ontologies are the model of choice to describe data semantics. Furthermore, standard modelling languages that developers are familiar with (e.g., BPEL or CDL) can be used to specify the interaction protocols and then automatically translate them to FSP using existing tools[15].
- *Dealing with encryption.* When encryption is enforced (e.g., Google Talk encrypts XMPP messages), the mediator cannot parse or modify these messages all the way between the initial sender and the ultimate receiver. Transparency cannot be ensured anymore. Instead, the user get involved and handles some of the translation tasks [113]. In the Google Talk case, the mediator uses a robot (bot) that the user adds to its contact list. The robot manages a set of commands, e.g., `IM <destinationID> <message>` to send a message `message` to user `destinationID`.

D.7 Related Work

The problem of mediating applications has been studied in different domains. Middleware solutions focus on providing abstraction and execution environments that enable interoperation by providing an abstract interface and exploiting reflection [54], by translating into a common intermediary protocol such as in the case of Enterprise Service Buses [85] or by proposing a domain-specific language to describe the translation logic and automatically generate the corresponding gateways [26]. However, these solutions require the developer to specify the translation to be made and hence to know both protocols in advance whereas in our approach, each protocol is independently specified and the translation is produced automatically. The Web Service Execution Environment (WSMX) performs the necessary translation on the basis of pre-defined mediation patterns. However, the composition of these patterns is not considered, and there is no guarantee that it will not lead to a deadlock. Vaculín *et al.* [111] devise a mediation approach for OWL-S processes. They first generate all requester paths, then find the appropriate mapping for each path by simulating the provider process. This approach deals only with client/server Web service interactions. It is not able to deal with the heterogeneity of instant messaging applications for example. Calvert and Lam [31] propose an approach to reason about the existence of a mediator by projecting both systems into a common sub-protocol. However, this common sub-protocol needs to be specified using an intuitive understanding of the protocols. In their seminal paper, Yellin and Strom [119] propose an algorithm for the automated synthesis of mediators based on predefined correspondences between messages. By considering the semantics of actions, we are able to infer the correspondences between messages automatically. Finally, Cavallaro *et al.* [33] also consider the semantics of data and relies on model checking to identify mapping scripts between interaction protocols automatically. However, they do not take into account the actual semantics of the operations. Moreover, they propose to perform the interface mapping beforehand so as to align the vocabulary of the processes, but many mappings may exist and should be considered during the generation of the mediator. Hence, even though there exists a significant amount of work to achieve interoperability, none of the existing approaches proposes to generate automatically mediators that are able to deal with both data and protocol mismatches.

[15] http://www.doc.ic.ac.uk/ltsa/bpel4ws/

D.8 Conclusion

Achieving interoperability between heterogeneous distributed applications without actually modifying their interfaces or behaviour is desirable and often necessary in today's pervasive systems. Mediators promote the seamless interconnection of distributed applications by performing the necessary translations between their messages and coordinating their behaviour. In this paper, we have presented a principled approach to the automated synthesis of mediators at runtime. We first infer mappings between application interfaces by reasoning about the semantics of their data and operations annotated using a domain-specific ontology. We then use these mappings to automatically synthesise a correct-by-construction mediator. This principled approach to generating mediators removes the need to develop *ad hoc* bridging solutions and fosters future-proof interoperability. We evaluated the approach using a case study involving heterogeneous instant messaging applications and showed that it can successfully ensure their interoperation.

Work in progress includes the definition of many-to-many operation mappings to manage a broader set of heterogeneous systems. We are also investigating the synthesis of mediators between more than a pair of networked applications. This is for example the case when IM conversations involve multiple users. Our work further integrates with complementary work ongoing within the CONNECT European project so as to develop a framework to support the interoperability lifecycle by using semantic technologies to synthesise mediators dynamically and ensure their evolution to respond efficiently to changes in the individual systems or in the ontology. A further direction is to consider improved modelling capabilities that take into account the probabilistic nature of systems and the uncertainties in the ontology. This would facilitate the construction of mediators where we have only partial knowledge about the system.

Bibliography

[1] R. Allen and D. Garlan. A formal basis for architectural connection. *ACM Trans. Softw. Eng. Methodol.*, 1997.

[2] M. Aranguren, S. Bechhofer, P. Lord, U. Sattler, and R. Stevens. Understanding and using the meaning of statements in a bio-ontology: recasting the gene ontology in OWL. *BMC bioinformatics*, 8(1):57, 2007.

[3] D. Ardagna, C. Ghezzi, and R. Mirandola. Model driven qos analyses of composed web services. In *ServiceWave*, pages 299–311, 2008.

[4] A. Artale, E. Franconi, N. Guarino, and L. Pazzi. Part-whole relations in object-centered systems: An overview. *Data Knowl. Eng.*, 20(3):347–383, 1996.

[5] M. Autili, P. Inverardi, A. Navarra, and M. Tivoli. SYNTHESIS: A tool for automatically assembling correct and distributed component-based systems. In *International Conference on Software Engineering, ICSE*, 2007.

[6] F. Baader, D. Calvanese, D. L. McGuinness, D. Nardi, and P. F. Patel-Schneider. *The Description Logic Handbook*. Cambridge University Press, 2003.

[7] L. Baresi, E. Di Nitto, and C. Ghezzi. Toward open-world software: Issue and challenges. *IEEE Computer*, 39(10):36–43, 2006.

[8] S. Ben Mokhtar. *Intergiciel Sémantique pour les Services de l'Informatique Diffuse*. PhD thesis, Université Pierre et Marie Curie - Paris VI, Dec. 2007.

[9] A. Bennaceur, G. S. Blair, F. Chauvel, G. Huang, N. Georgantas, P. Grace, F. Howar, P. Inverardi, V. Issarny, M. Paolucci, A. Pathak, R. Spalazzese, B. Steffen, and B. Souville. Towards an architecture for runtime interoperability. In *ISoLA (2)*, pages 206–220, 2010.

[10] A. Bennaceur, V. Issarny, J. Richard, M. Alessandro, S. Romina, and D. Sykes. Automatic service categorisation through machine learning in emergent middleware. In *Software Technologies Concertation on Formal Methods for Components and Objects, FMCO*, 2011.

[11] A. Bennaceur, V. Issarny, R. Spalazzese, and S. Tyagi. Achieving interoperability through semantics-based technologies: The instant messaging case. In *11th International Semantic Web Conference, ISWC*, 2012.

[12] A. Bennaceur, V. Issarny, D. Sykes, F. Howar, M. Isberner, B. Steffen, R. Johansson, and A. Moschitti. Machine learning for emergent middleware. In *Proc. of the Joint workshop on Intelligent Methods for Soft. System Eng., JIMSE*, 2012. to appear.

[13] M. Bernardo. Twotowers 5.1 user manual, 2006.

[14] M. Bernardo and M. Bravetti. Performance measure sensitive congruences for markovian process algebras. *Theor. Comput. Sci.*, 290(1):117–160, 2003.

[15] M. Bernardo, P. Ciancarini, and L. Donatiello. Architecting families of software systems with process algebras. *ACM TOSEM*, 11:386–426, 2002.

[16] M. Bersani, L. Cavallaro, A. Frigeri, M. Pradella, and M. Rossi. SMT-based Verification of LTL Specification with Integer Constraints and its Application to Runtime Checking of Service Substitutability. In *Software Engineering and Formal Methods (SEFM), 2010 8th IEEE International Conference on*, pages 244–254. IEEE, 2010.

[17] A. Bertolino, A. Calabro', F. Di Giandomenico, N. Nostro, P. Inverardi, and R. Spalazzese. On-the-fly dependable mediation between heterogeneous networked systems. In *ICSOFT 2011, CCIS 303*, pages 20–37. Springer-Verlag Berlin Heidelberg, 2012, 2012.

[18] A. Bertolino, A. Calabrò, M. Merten, and B. Steffen. Never-stop learning: Continuous validation of learned models for evolving systems through monitoring. *ERCIM News*, 2012(88), 2012.

[19] G. S. Blair, A. Bennaceur, N. Georgantas, P. Grace, V. Issarny, V. Nundloll, and M. Paolucci. The role of ontologies in emergent middleware: Supporting interoperability in complex distributed systems. In *Middleware'11*, 2011.

[20] A. Borgida. From type systems to knowledge representation: Natural semantics specifications for description logics. *Int. J. Cooperative Inf. Syst.*, 1(1):93–126, 1992.

[21] A. Borgida. How knowledge representation meets software engineering (and often databases). *Automated Soft. Eng.*, 14(4), 2007.

[22] A. Borgida and P. T. Devanbu. Adding more "DL" to IDL: Towards more knowledgeable component interoperability. In *International Conference on Software Engineering, ICSE*, pages 378–387, 1999.

[23] N. Borisov, D. J. Brumley, and H. J. Wang. A generic application-level protocol analyzer and its language. In *In 14th Annual Network Distributed System Security Symposium*, 2007.

[24] A. Bracciali, A. Brogi, and C. Canal. A formal approach to component adaptation. *Journal of Syst. and Softw.*, 2005.

[25] D. Brand and P. Zafiropulo. On communicating finite-state machines. *Journal of the ACM*, 30(2):323–342, 1983.

[26] Y.-D. Bromberg, P. Grace, and L. Réveillère. Starlink: Runtime interoperability between heterogeneous middleware protocols. In *Proc. ICDCS*, 2011.

[27] Y.-D. Bromberg, P. Grace, L. Réveillère, and G. S. Blair. Bridging the interoperability gap: Overcoming combined application and middleware heterogeneity. In *Middleware*, pages 390–409, 2011.

[28] Y.-D. Bromberg, L. Réveillère, J. L. Lawall, and G. Muller. Automatic generation of network protocol gateways. In *Middleware*, 2009.

[29] L. Burgy, L. Reveillere, J. Lawall, and G. Muller. Zebu: A language-based approach for network protocol message processing. *IEEE Trans. on Soft. Eng.*, 37(4):575 –591, 2011.

[30] C. Calero, F. Ruiz, and M. Piattini. *Ontologies for Software Engineering and Software Technology*. Springer-Verlag, 2006.

[31] K. L. Calvert and S. S. Lam. Formal methods for protocol conversion. *IEEE Journal on Selected Areas in Comm.*, 1990.

[32] J. Cámara, J. A. Martín, G. Salaün, J. Cubo, M. Ouederni, C. Canal, and E. Pimentel. ITACA: An integrated toolbox for the automatic composition and adaptation of web services. In *International Conference on Software Engineering, ICSE*, 2009.

[33] L. Cavallaro, E. D. Nitto, and M. Pradella. An automatic approach to enable replacement of conversational services. In *ICSOC/ServiceWave*, 2009.

[34] H. Chang, L. Mariani, and M. Pezzè. In-field healing of integration problems with cots components. In *International Conference on Software Engineering, ICSE*, pages 166–176, 2009.

[35] D. A. Chappell. *Enterprise Service Bus*. O'Reilly, 2004.

[36] E. Cimpian and A. Mocan. WSMX process mediation based on choreographies. In *Proc. of Business Process Management Workshop*, 2005.

[37] E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. *ACM Trans. Program. Lang. Syst.*, 1994.

[38] V. Cortellessa, M. De Sanctis, A. Di Marco, and C. Trubiani. Enabling Performance Antipatterns to arise from an ADL-based Software Architecture. In *WICSA/ECSA 2012*, Helsinki, Finland, August, 2012., 2012.

[39] V. Cortellessa, A. Di Marco, and P. Inverardi. *Model-Based Software Performance Analysis*. Springer, 2011.

[40] M. d'Aquin and N. F. Noy. Where to publish and find ontologies? a survey of ontology libraries. *J. Web Sem.*, 11:96–111, 2012.

[41] K. Dentler, R. Cornet, A. ten Teije, and N. de Keizer. Comparison of reasoners for large ontologies in the OWL 2 EL profile. *Semantic Web*, 2(2):71–87, 2011.

[42] A. Di Marco, C. Pompilio, A. Bertolino, A. Calabrò, F. Lonetti, and A. Sabetta. Yet another meta-model to specify non-functional properties. In *QASBA*, pages 9–16, 2011.

[43] J. S. Dong. From semantic web to expressive software specifications: a modeling languages spectrum. In *International Conference on Software Engineering, ICSE*, 2006.

[44] D. Fensel, H. Lausen, A. Polleres, J. d. Bruijn, M. Stollberg, D. Roman, and J. Domingue. *Enabling Semantic Web Services: The Web Service Modeling Ontology*. Springer-Verlag, 2006.

[45] R. Fielding. *Architectural styles and the design of network-based software architectures*. PhD thesis, University of California, 2000.

[46] J. François, H. J. Abdelnur, R. State, and O. Festor. Semi-supervised fingerprinting of protocol messages. In *CISIS'10*, 2010.

[47] T. Franz and S. Staab. SAM: Semantics aware instant messaging for the networked semantic desktop. In *Proc. International Sem. Web Conf. Workshops*, 2005.

[48] A. Gangemi, N. Guarino, C. Masolo, A. Oltramari, and L. Schneider. Sweetening ontologies with DOLCE. In *EKAW*, pages 166–181, 2002.

[49] D. Garlan, R. Allen, and J. Ockerbloom. Architectural mismatch or why it's hard to build systems out of existing parts. In *International Conference on Software Engineering, ICSE*, 1995.

[50] D. Garlan, R. Allen, and J. Ockerbloom. Architectural mismatch: Why reuse is still so hard. *IEEE Software*, 2009.

[51] F. D. Giandomenico, P. Inverardi, N. Nostro, and R. Spalazzese. Enhanced connectors synthesis to address functional, performance, and dependability aspects. Technical report, 2012.

[52] C. Gierds, A. J. Mooij, and K. Wolf. Reducing adapter synthesis to controller synthesis. *IEEE T. Services Computing*, 5(1):72–85, 2012.

[53] J. Golbeck and M. Rothstein. Linking social networks on the web with foaf: A semantic web case study. In *AAAI*, pages 1138–1143, 2008.

[54] P. Grace, G. S. Blair, and S. Samuel. ReMMoC: A reflective middleware to support mobile client interoperability. In *CoopIS/DOA/ODBASE*, 2003.

[55] N. Guarino. Helping people (and machines) understanding each other: The role of formal ontology. In *CoopIS/DOA/ODBASE (1)*, page 599, 2004.

[56] S. Heiler. Semantic interoperability. *ACM Comput. Surv.*, 1995.

[57] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[58] C. A. R. Hoare. Process algebra: A unifying approach. In *25 Years Comm. Seq. Processes*, 2004.

[59] P. Inverardi, V. Issarny, and R. Spalazzese. A theory of mediators for eternal connectors. In *4th International Symposium on Leveraging Applications of Formal Methods, Verification, and Validation, ISoLA*, pages 236–250, 2010.

[60] P. Inverardi, R. Spalazzese, and M. Tivoli. Application-layer connector synthesis. In M. Bernardo and V. Issarny, editors, *SFM-11: 11th International School on Formal Methods for the Design of Computer, Communication and Software Systems – Connectors for Eternal Networked Software Systems*. Springer Verlag, 2011.

[61] V. Issarny, A. Bennaceur, and Y.-D. Bromberg. Middleware-layer connector synthesis: Beyond state of the art in middleware interoperability. In M. Bernardo and V. Issarny, editors, *SFM-11: 11th International School on Formal Methods for the Design of Computer, Communication and Software Systems – Connectors for Eternal Networked Software Systems*. Springer Verlag, 2011.

[62] J. Jiang, S. Zhang, P. Gong, and Z. Hong. Message dependency-based adaptation of services. In *APSCC*, pages 442–449, 2011.

[63] J. Jiang, S. Zhang, P. Gong, and Z. Hong. Service adaptation at message level. In *SERVICES*, pages 87–88, 2011.

[64] U. Junker and D. Mailharro. The logic of ilog (j) configurator: Combining constraint programming with a description logic. In *proceedings of Workshop on Configuration, IJCAI*, volume 3, pages 13–20, 2003.

[65] R. M. Karp. Reducibility among combinatorial problems. In R. E. Miller and J. W. Thatcher, editors, *Proc. of a Symposium on the Complexity of Computer Computations*, pages 85–103, New York, NY, 1972. IBM Research Symposia Series, Plenum Press.

[66] R. M. Keller. Formal verification of parallel programs. *Communications of the ACM*, 19(7):371–384, 1976.

[67] D. Kimelman, M. Kimelman, D. Mandelin, and D. M. Yellin. Bayesian approaches to matching architectural diagrams. *IEEE Trans. Software Eng.*, 36(2):248–274, 2010.

[68] D. Konstantas, J.-P. Bourrires, M. Lonard, and N. Boudjlida. *Interoperability of enterprise software and applications*. Springer-Verlag, 2006.

[69] J. Kopecký, T. Vitvar, C. Bournez, and J. Farrell. SAWSDL: Semantic annotations for WSDL and XML schema. *IEEE Internet Computing*, 11(6):60–67, 2007.

[70] H. Koziolek. Performance evaluation of component-based software systems: A survey. *Perform. Eval.*, 67(8):634–658, 2010.

[71] I. Krka, Y. Brun, D. Popescu, J. Garcia, and N. Medvidovic. Using dynamic execution traces and program invariants to enhance behavioral model inference. In *International Conference on Software Engineering, ICSE (2)*, pages 179–182, 2010.

[72] F. Laburthe. Constraints over ontologies. In *CP*, pages 878–882, 2003.

[73] S. S. Lam. Protocol conversion. *IEEE Transaction Software Engineering*, 1988.

[74] S. S. Lam and A. U. Shankar. Protocol verification via projections. *IEEE Trans. Software Eng.*, 10(4):325–342, 1984.

[75] L. Lefort, K. Taylor, and D. Ratcliffe. An empirical comparison of scalable part-whole ontology engineering patterns. *Expert Systems*, 25(3):295–313, 2008.

[76] B. Liskov and J. M. Wing. A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst.*, 16(6):1811–1841, 1994.

[77] D. Lorenzoli, L. Mariani, and M. Pezzè. Automatic generation of software behavioral models. In *International Conference on Software Engineering, ICSE*, pages 501–510, 2008.

[78] J. Magee and J. Kramer. *Concurrency : State models and Java programs*. Hoboken (N.J.) : Wiley, 2006.

[79] M. W. Maier. Integrated modeling: A unified approach to system engineering. *Journal of Systems and Software*, 32(2):101–119, 1996.

[80] S. Marquis, T. R. Dean, and S. Knight. SCL: a language for security testing of network applications. In *CASCON'06*, 2005.

[81] D. L. Martin, M. H. Burstein, D. V. McDermott, S. A. McIlraith, M. Paolucci, K. P. Sycara, D. L. McGuinness, E. Sirin, and N. Srinivasan. Bringing semantics to web services with owl-s. In *Proc. if the World Wide Web conference, WWW'07*, pages 243–277, 2007.

[82] J. A. Martín and E. Pimentel. Automatic generation of adaptation contracts. *Electronic Notes in Theoretical Computer Science*, 229(2):115–131, 2009.

[83] R. Mateescu, P. Poizat, and G. Salaun. Adaptation of service protocols using process algebra and on-the-fly reduction techniques. *IEEE Transactions on Software Engineering*, 99(PrePrints), 2011.

[84] R. Mateescu, P. Poizat, and G. Salaun. Adaptation of service protocols using process algebra and on-the-fly reduction techniques. *IEEE Trans. on Soft. Eng.*, 38(4), 2012.

[85] F. Menge. Enterprise Service Bus. In *Proc. of the Free and open source soft. conf.*, 2007.

[86] M. Merten, F. Howar, B. Steffen, P. Pellicione, and M. Tivoli. Automated inference of models for black box systems based on interface descriptions. In T. Margaria and B. Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change*, volume 7609 of *Lecture Notes in Computer Science*, pages 79–96. Springer Berlin / Heidelberg, 2012.

[87] S. B. Mokhtar, A. Kaul, N. Georgantas, and V. Issarny. Efficient semantic service discovery in pervasive computing environments. In *Middleware*, 2006.

[88] P. Mork, L. Seligman, A. Rosenthal, J. Korb, and C. Wolf. The harmony integration workbench. *Journal on Data Semantics*, 11, 2008.

[89] M. A. Motoyama and G. Varghese. CrossTalk: scalably interconnecting instant messaging networks. In *Proc. ACM Workshop on Online Social Networks*, 2009.

[90] J. Nakazawa, H. Tokuda, W. K. Edwards, and U. Ramachandran. A bridging framework for universal interoperability in pervasive systems. In *Proceedings of ICDCS*, 2006.

[91] H. R. M. Nezhad, G. Y. Xu, and B. Benatallah. Protocol-aware matching of web service interfaces for adapter development. In *WWW*, 2010.

[92] Nielsen. *Games Dominate America's Growing Appetite for Mobile Apps*, 2010.

[93] I. Niles and A. Pease. Towards a standard upper ontology. In *FOIS*, pages 2–9, 2001.

[94] N. Oldham, C. Thomas, A. P. Sheth, and K. Verma. METEOR-S web service annotation framework with machine learning classification. In *SWSWPC*, pages 137–146, 2004.

[95] Z. J. Oster, G. R. Santhanam, and S. Basu. Identifying optimal composite services by decomposing the service composition problem. In *ICWS*, pages 267–274, 2011.

[96] L. Padovani. Contract-based discovery of web services modulo simple orchestrators. *Theor. Comput. Sci.*, 411(37):3328–3347, 2010.

[97] M. Paolucci, T. Kawamura, T. R. Payne, and K. P. Sycara. Semantic matching of web services capabilities. In *International Semantic Web Conference, ISWC*, 2002.

[98] C. Petrie, T. Margaria, H. Lausen, and M. Zaremba. *Semantic Web Services Challenge: Results from the First Year*, volume 8. Springer, 2008.

[99] R. G. Raskin and M. J. Pan. Knowledge representation in the semantic web for earth and environmental terminology (SWEET). *Computers & Geosciences*, 31(9):1119–1125, 2005.

[100] F. Rossi, P. Van Beek, and T. Walsh. *Handbook of constraint programming*, volume 35. Elsevier Science, 2006.

[101] N. Shadbolt, T. Berners-Lee, and W. Hall. The semantic web revisited. *IEEE Intelligent Systems*, 21(3):96–101, 2006.

[102] M. Shaw. Architectural issues in software reuse: It's not just the functionality, it's the packaging. In *SSR*, 1995.

[103] P. Shvaiko and J. Euzenat. A survey of schema-based matching approaches. *J. Data Semantics IV*, pages 146–171, 2005.

[104] R. Spalazzese and P. Inverardi. Mediating connector patterns for components interoperability. In *ECSA*, 2010.

[105] R. Spalazzese, P. Inverardi, and V. Issarny. Towards a formalization of mediating connectors for on the fly interoperability. In *WICSA/ECSA*, 2009.

[106] B. Spitznagel and D. Garlan. A compositional formalization of connector wrappers. In *International Conference on Software Engineering, ICSE*, 2003.

[107] W. J. Stewart. *Introduction to the numerical solution of Markov Chains*. Princeton University Press, 1994.

[108] The Radicati Group. *Instant Messaging Market 10-14*, 2010.

[109] M. Tivoli, P. Fradet, A. Girault, and G. Gößler. Adaptor synthesis for real-time components. In *TACAS*, pages 185–200, 2007.

[110] R. Vaculín, R. Neruda, and K. P. Sycara. An agent for asymmetric process mediation in open environments. In *SOCASE*, volume 5006 of *Lecture Notes in Computer Science*, pages 104–117. Springer, 2008.

[111] R. Vaculín, R. Neruda, and K. P. Sycara. The process mediation framework for semantic web services. *International Journal of Agent-Oriented Software Engineering, IJAOSE*, 3(1):27–58, 2009.

[112] R. Vaculín and K. P. Sycara. Towards automatic mediation of owl-s process models. In *IEEE nternational Conference on Web Services, ICWS, International Conference on Web Services*, pages 1032–1039, 2007.

[113] C. Vassilakis and C. Kareliotis. A framework for adaptation in secure web services. In *Medi. Conf. on Info. Syst.*, 2009.

[114] P. Welch. *The Statistical Analysis of Simulation Results*. Academic Press, 1983.

[115] G. Wiederhold. Mediators in the architecture of future information systems. *Computer*, 25(3):38–49, 1992.

[116] G. Wiederhold. Interoperation, mediation, and ontologies. In *Proc. of the Fifth International Symposium on Generation Computer Systems Workshop on Heterogeneous Cooperative Knowledge-Bases*, pages 33–48. Citeseer, 1994.

[117] S. K. Williams, S. A. Battle, and J. E. Cuadrado. Protocol mediation for adaptation in semantic web services. In *ESWC*, pages 635–649, 2006.

[118] Z. Wu, K. Gomadam, A. Ranabahu, A. P. Sheth, and J. A. Miller. Automatic composition of semantic web services using process mediation. In *ICEIS (4)*, 2007.

[119] D. M. Yellin and R. E. Strom. Protocol specifications and component adaptors. *ACM Trans. Program. Lang. Syst.*, 1997.